

Real time operating system

Free RTOS & CMSIS RTOS V2

STM 32

4 types of real time operating system



01 – Hard Real time operating system



02 – Soft real time operating system



03 – Firm Real-time operating system



04 – Deterministic Real-time operating system

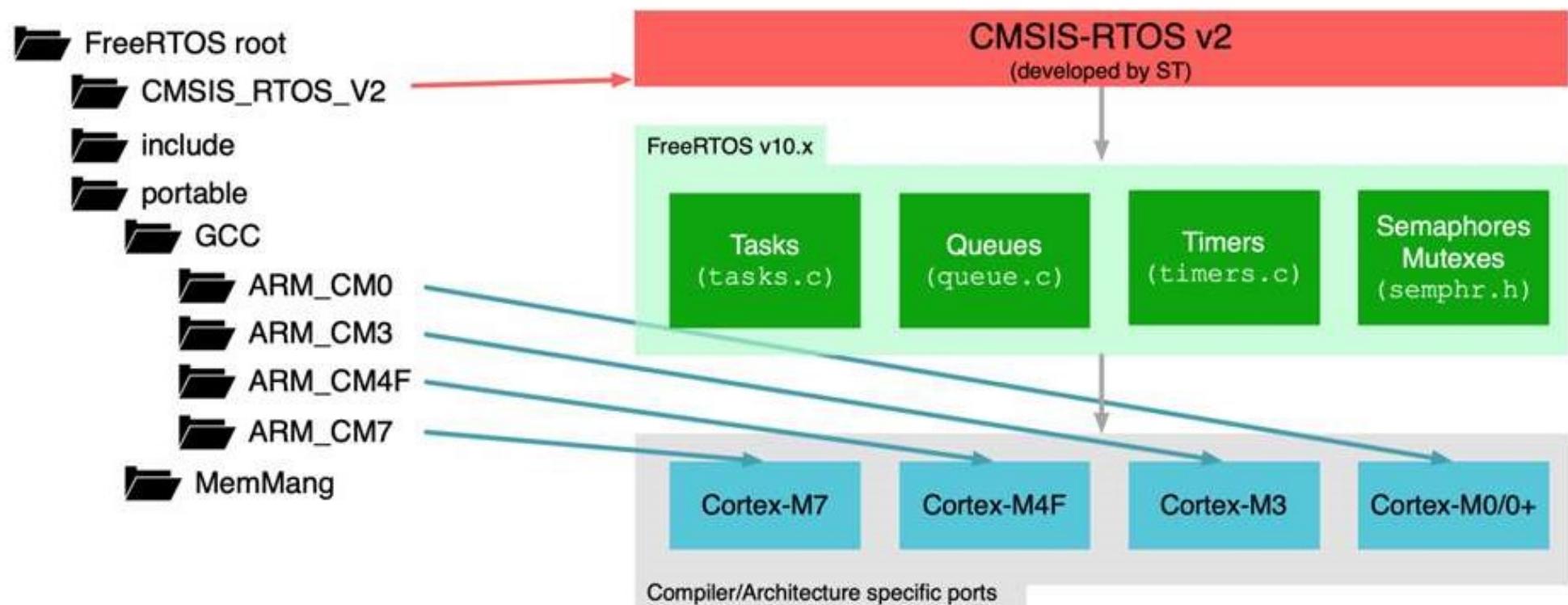


Figure 23.4: The FreeRTOS source tree organization in the CubeHAL

All available tasks appear to be executing ...

Task 1 Executing



Task 2 Executing

Task 3 Executing

Time

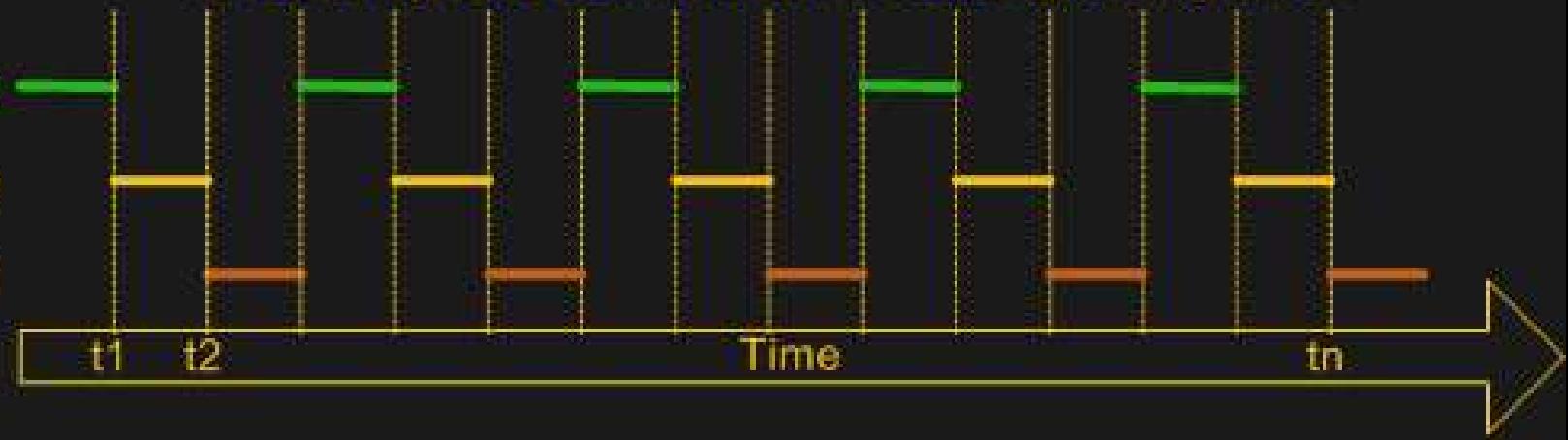
tn

... but only one task is ever executing at any time.

Task 1 Executing

Task 2 Executing

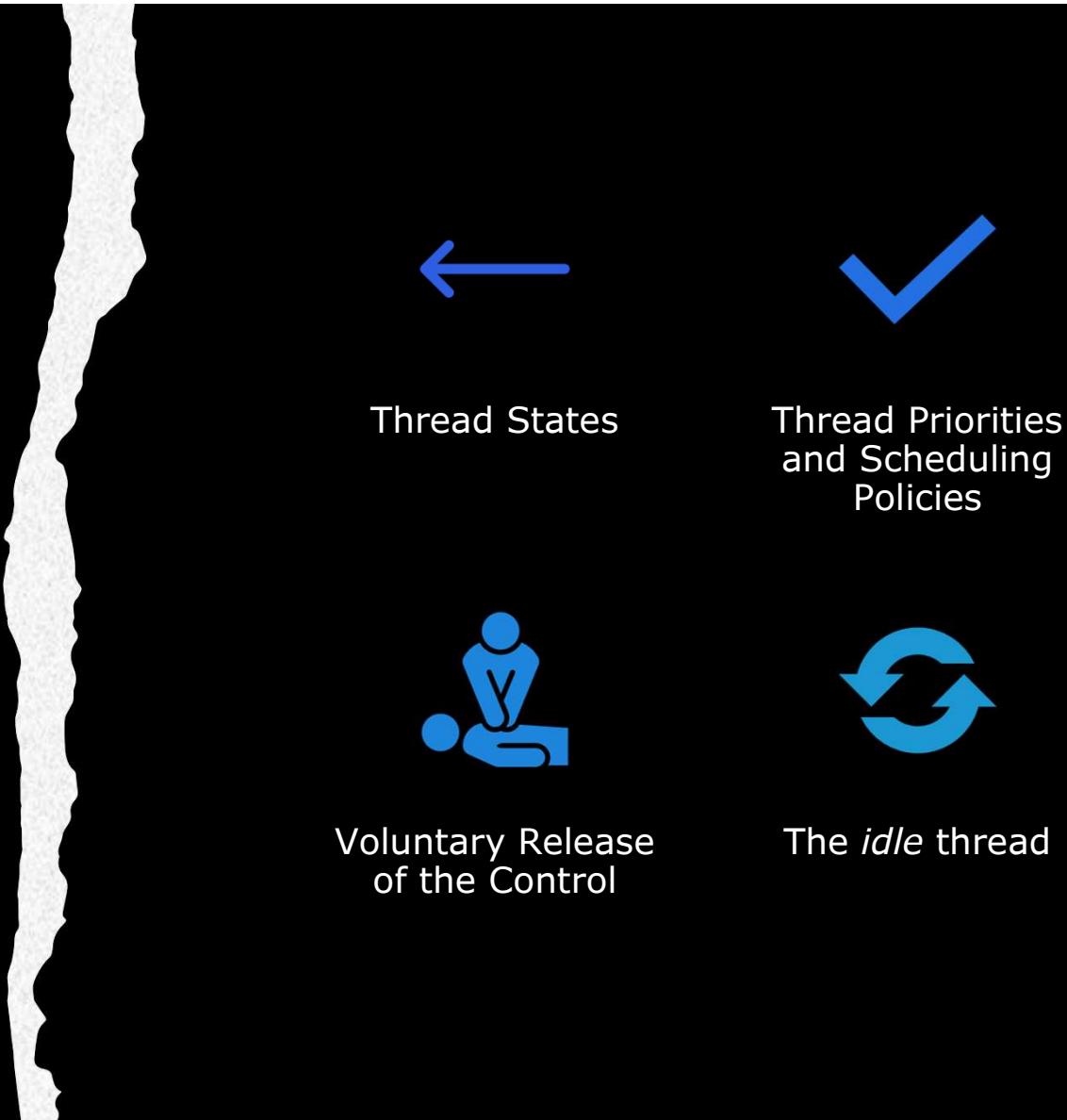
Task 3 Executing

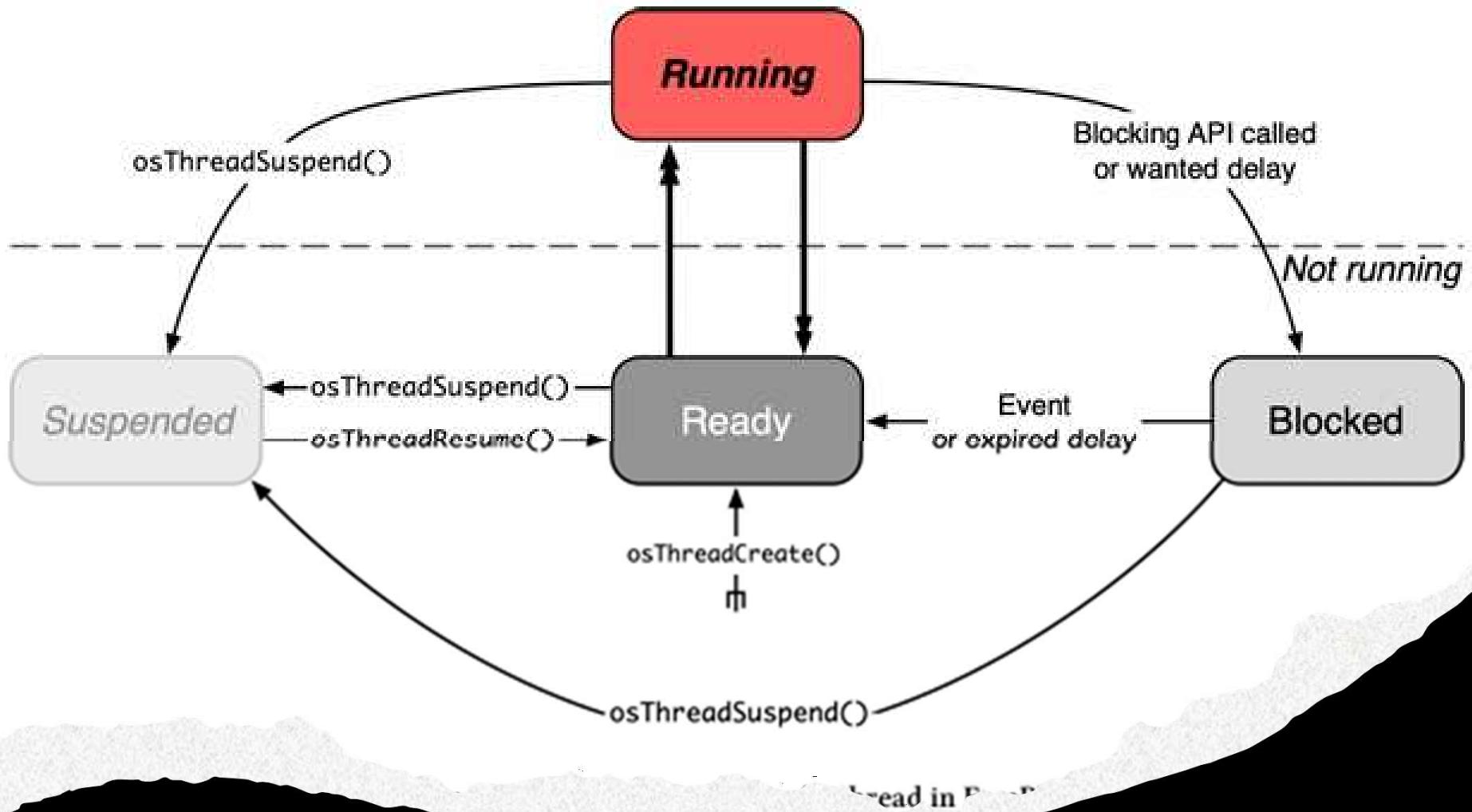


Time

tn

Thread Management





Priority level	Value	Description
<code>osPriorityNone</code>	0	No priority (not initialized).
<code>osPriorityIdle</code>	1	Priority: <i>idle</i> - Reserved for Idle thread.
<code>osPriorityLow</code>	8	Priority: <i>low</i>
<code>osPriorityLow[1..7]</code>	8+[1..7]	Priority: <i>low</i> + 1..7
<code>osPriorityBelowNormal</code>	16	Priority: <i>below normal</i>
<code>osPriorityBelowNormal[1..7]</code>	16+1	Priority: <i>below normal</i> + 1..7
<code>osPriorityNormal</code>	24	Priority: <i>normal</i>
<code>osPriorityNormal[1..7]</code>	24+1	Priority: <i>normal</i> + 1..7
<code>osPriorityAboveNormal</code>	32	Priority: <i>above normal</i>
<code>osPriorityAboveNormal[1..7]</code>	32+1	Priority: <i>above normal</i> + 1
<code>osPriorityHigh</code>	40	Priority: <i>high</i>
<code>osPriorityHigh[1..7]</code>	40+1	Priority: <i>high</i> + 1
<code>osPriorityRealtime</code>	48	Priority: <i>realtime</i>
<code>osPriorityRealtime[1..7]</code>	48+1	Priority: <i>realtime</i> + 1
<code>osPriorityISR</code>	56	Priority: <i>ISR</i> - Reserved for ISR deferred thread
<code>osPriorityError</code>	-1	System cannot determine priority or illegal priority
<code>osPriorityReserved</code>	0x7FFFFFFF	Prevents enum down-size compiler optimization

Variables Convention

Variables of type '*unsigned long*' are prefixed with '*ul*', where the '*u*' denotes '*unsigned*' and the '*l*' denotes '*long*'.

Variables of type '*unsigned short*' are prefixed with '*us*', where the '*u*' denotes '*unsigned*' and the '*s*' denotes '*short*'

Variables of type '*unsigned char*' are prefixed with '*uc*', where the '*u*' denotes '*unsigned*' and the '*c*' denotes '*char*'.



Variables of **non stdint** types are prefixed with '*x*'

Unsigned variables of non stdint types have an additional prefix '*u*'

Enumerated variables are prefixed with '*e*'

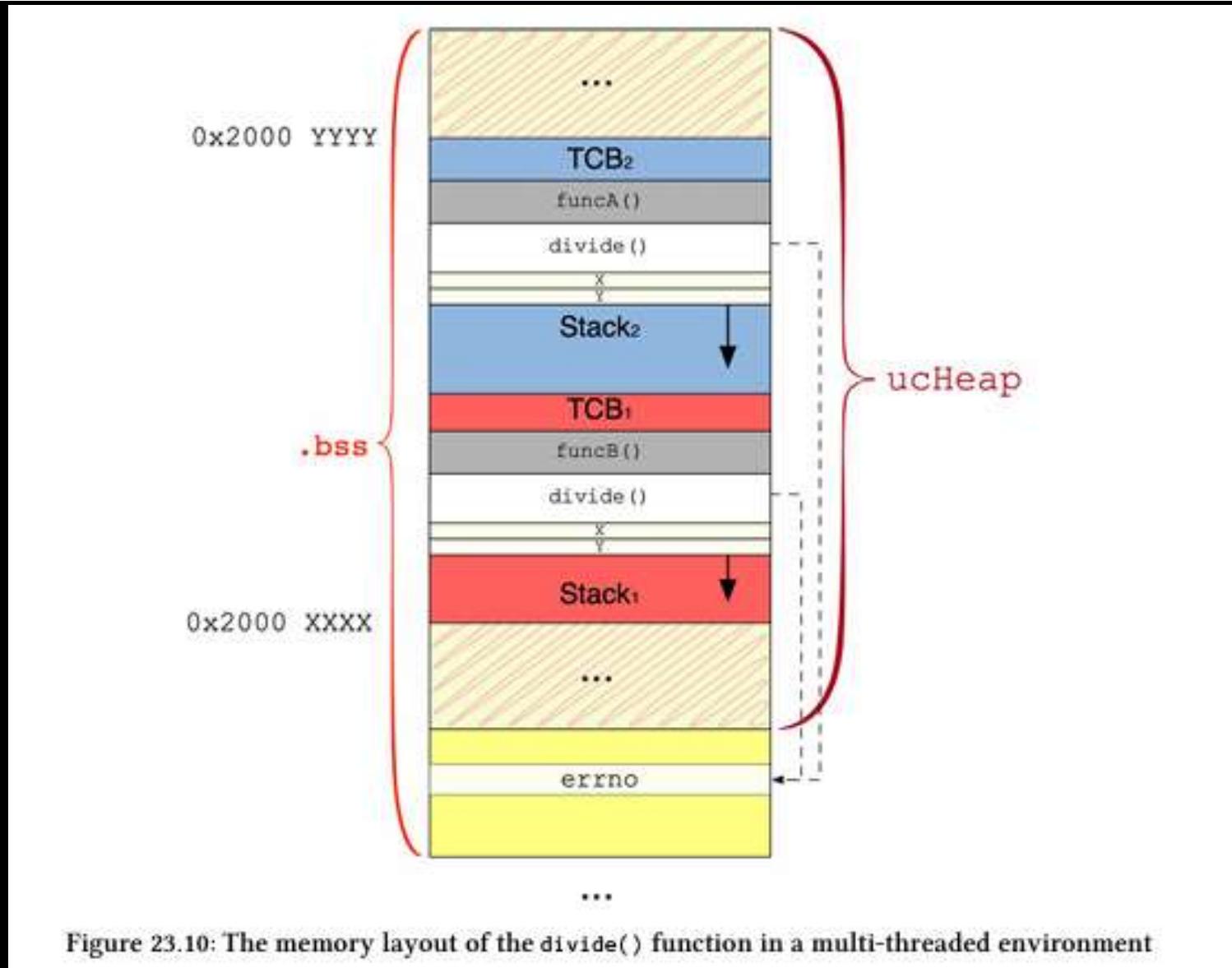


Figure 23.10: The memory layout of the `divide()` function in a multi-threaded environment



MULTITHREADING

THREADS ARE NOT GOING TO SYNCHRONIZE THEMSELVES

Figure 23.8: What usually happens when the number of thread increases too much

- Generic strategies:
 - **Strategy #1:** user-defined solution for handling thread safety.
 - **Strategy #2:** allows lock usage from interrupts. This implementation ensures thread safety by disabling all interrupts during, for instance, calls to `malloc()`.
 - **Strategy #3:** denies lock usage from interrupts. This implementation assumes single-thread execution and denies any attempt to take a lock from ISR context.
- FreeRTOS-based strategies:
 - **Strategy #4:** allows lock usage from interrupts. Implemented using FreeRTOS locks. This implementation ensures thread safety by entering RTOS ISR capable critical sections during, for instance, calls to `malloc()`. This implies that thread safety is achieved by disabling low-priority interrupts and task switching. High-priority interrupts are however not safe.
 - **Strategy #5:** denies lock usage from interrupts. Implemented using FreeRTOS locks. This implementation ensures thread safety by suspending all tasks during, for instance, calls to `malloc()`.

The screenshot shows the CubeMX software interface with the following configuration:

- Pinout & Configuration**: Selected.
- Clock Configuration**: Unselected.
- Project Manager**: Unselected.

Code Generator section:

- Toolchain / IDE**: STM32CubeIDE
- Generate Under Root

Advanced Settings section:

- Linker Settings**:
 - Minimum Heap Size: 0x200
 - Minimum Stack Size: 0x400
- Thread-safe Settings**:
 - Cortex-M4NS
 - Enable multi-threaded support
- Thread-safe Locking Strategy**:
 - Generic Strategy #2 - Allow lock usage from interrupts
 - Default – Mapping suitable strategy depending on RTOS selection.
 - Generic Strategy #1 - Custom implementation
 - Generic Strategy #2 - Allow lock usage from interrupts
 - Generic Strategy #3 - Deny lock usage from interrupts
 - FreeRTOS Strategy #4 - Allow lock usage from interrupts
 - FreeRTOS Strategy #5 - Deny lock usage from interrupts

Figure 23.14: How to select Thread-Safe Locking Strategy in CubeMX

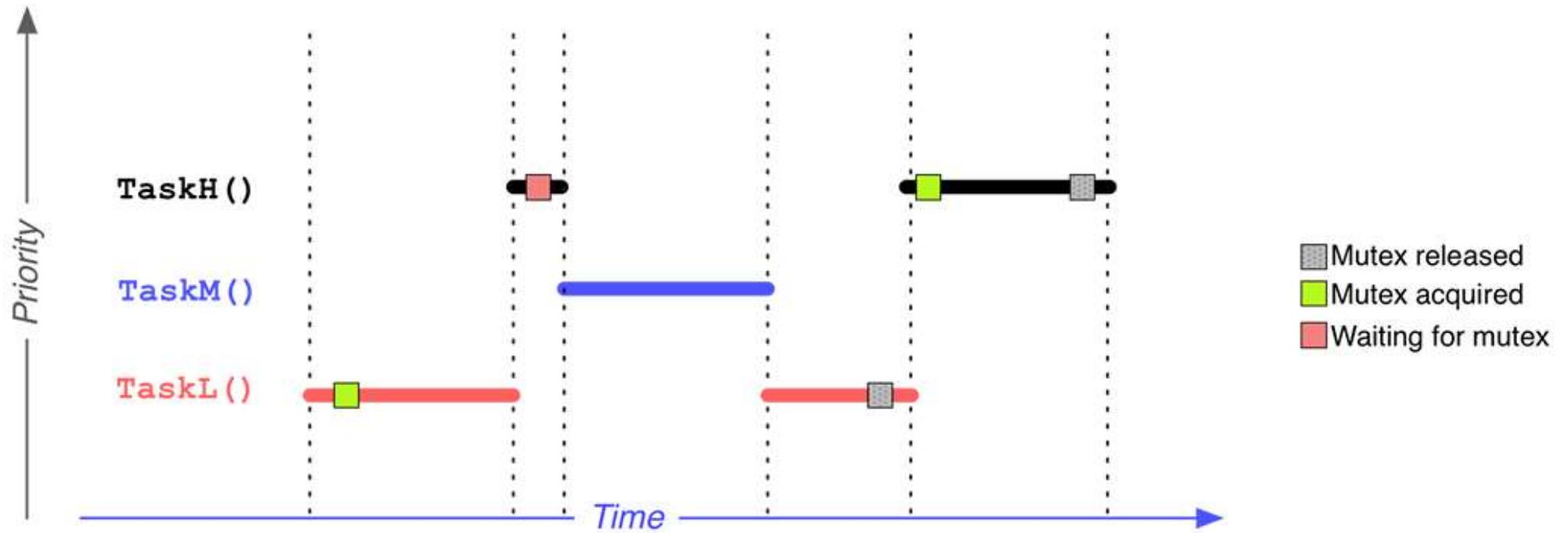


Figure 23.19: The diagram schematizes the *priority inversion* problem

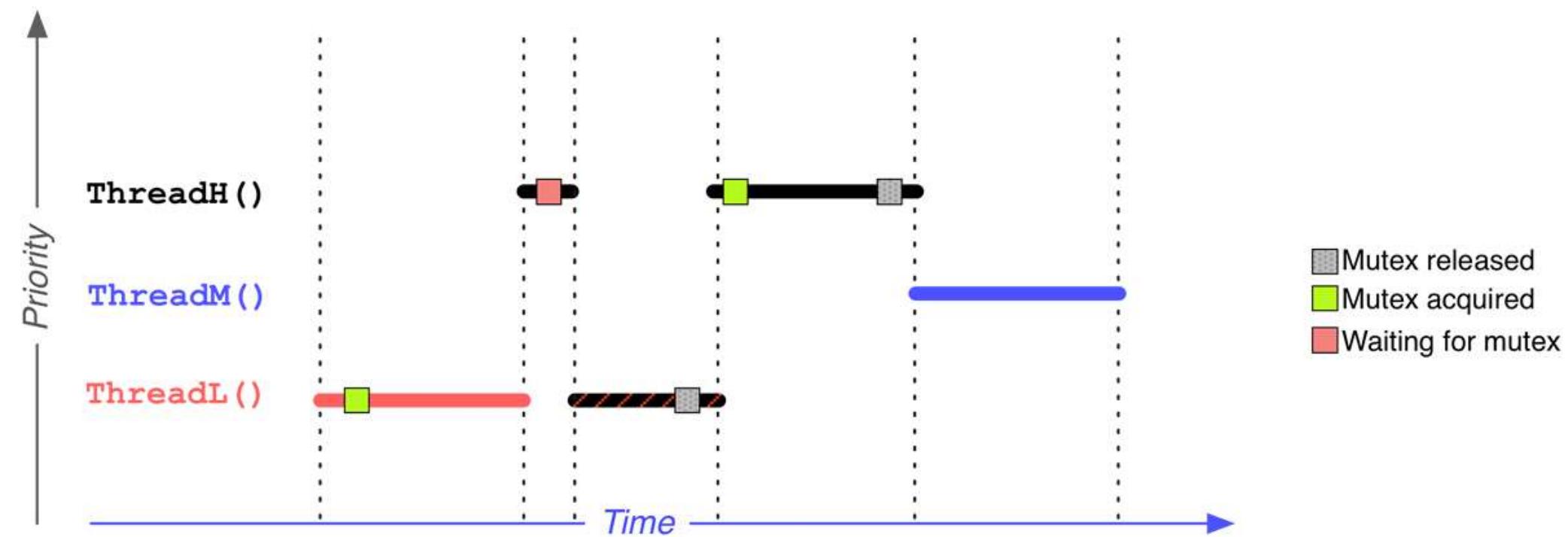
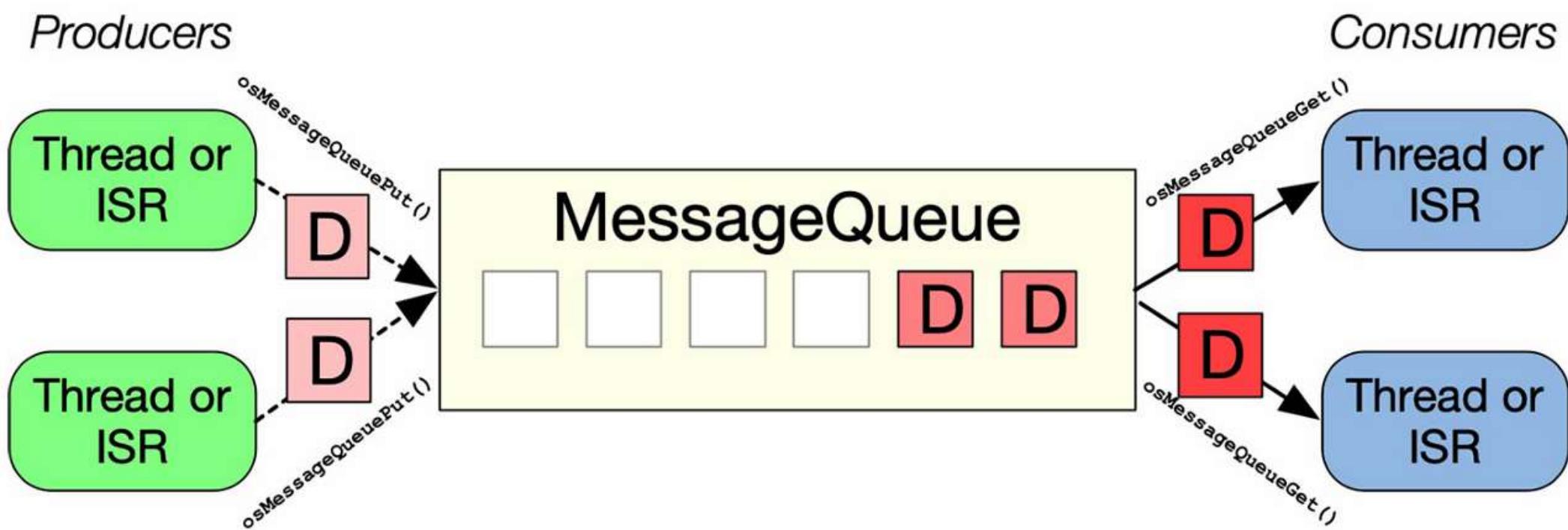
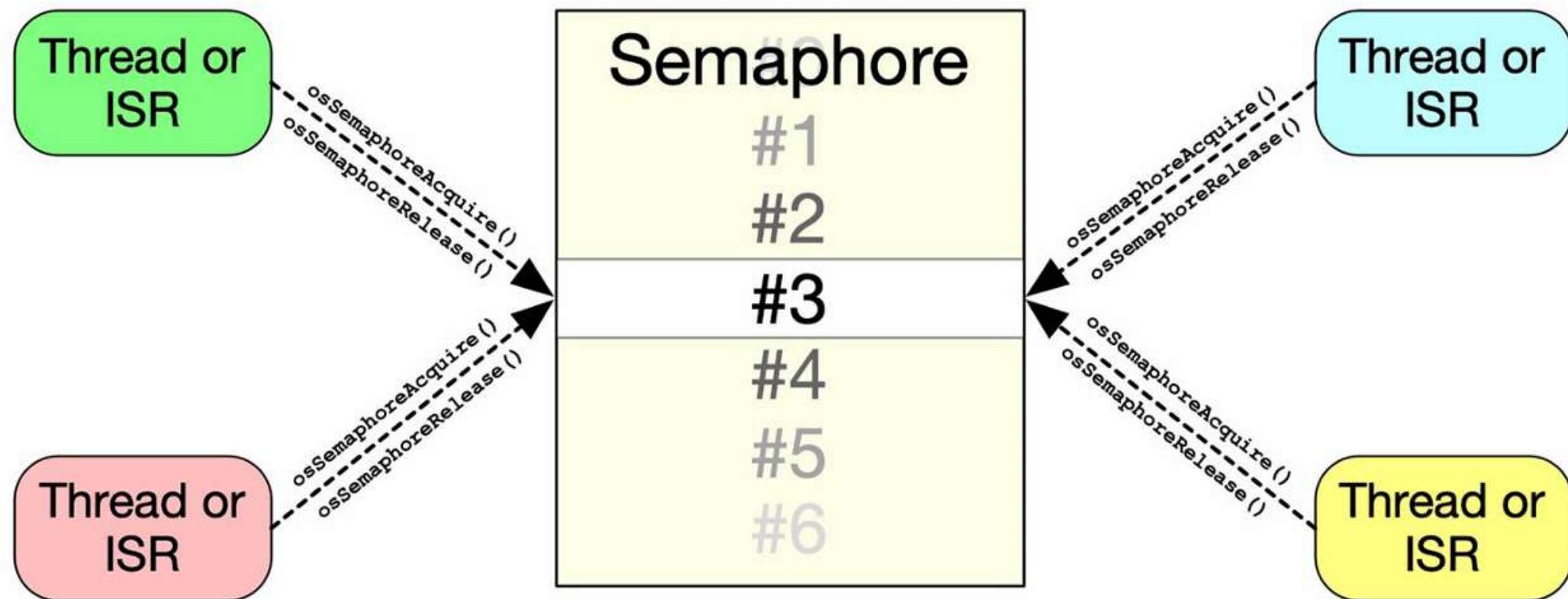


Figure 23.20: How the *priority inversion* problem is addressed by temporary increasing the priority of ThreadL

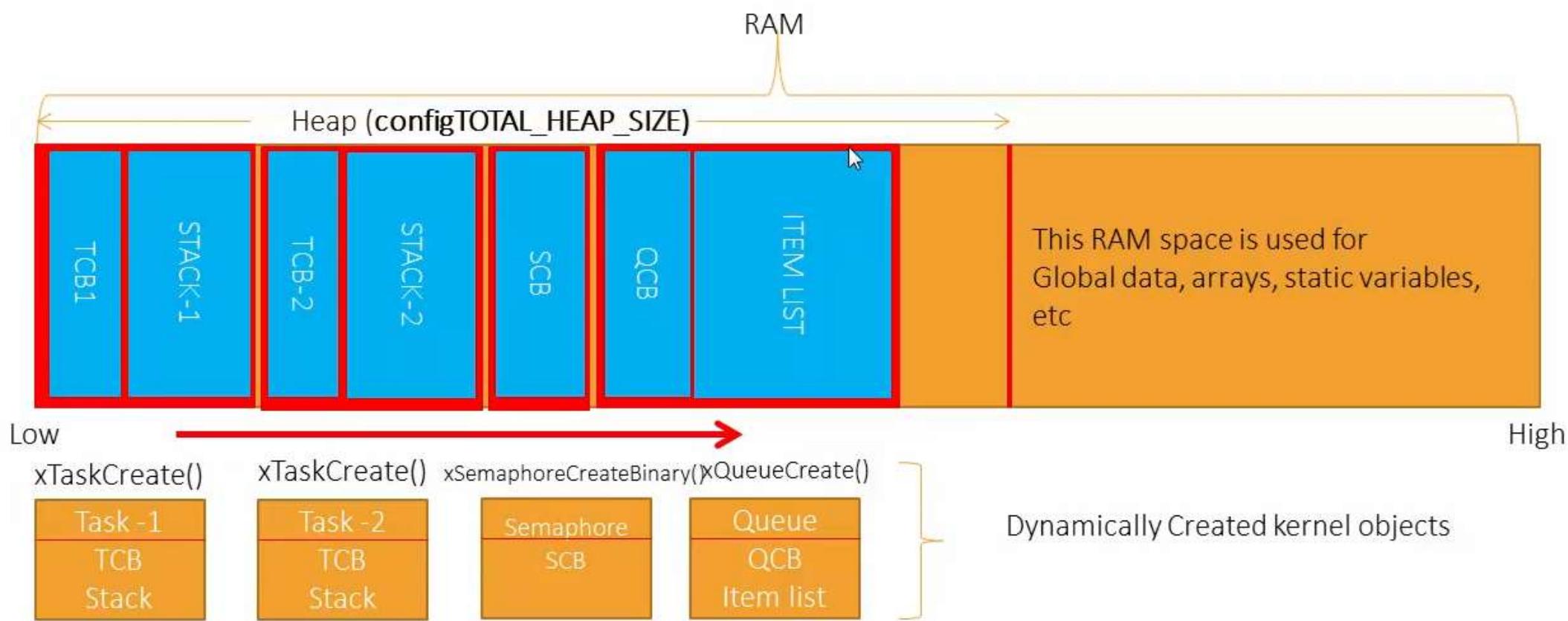




Memory Management in FREE RTOS

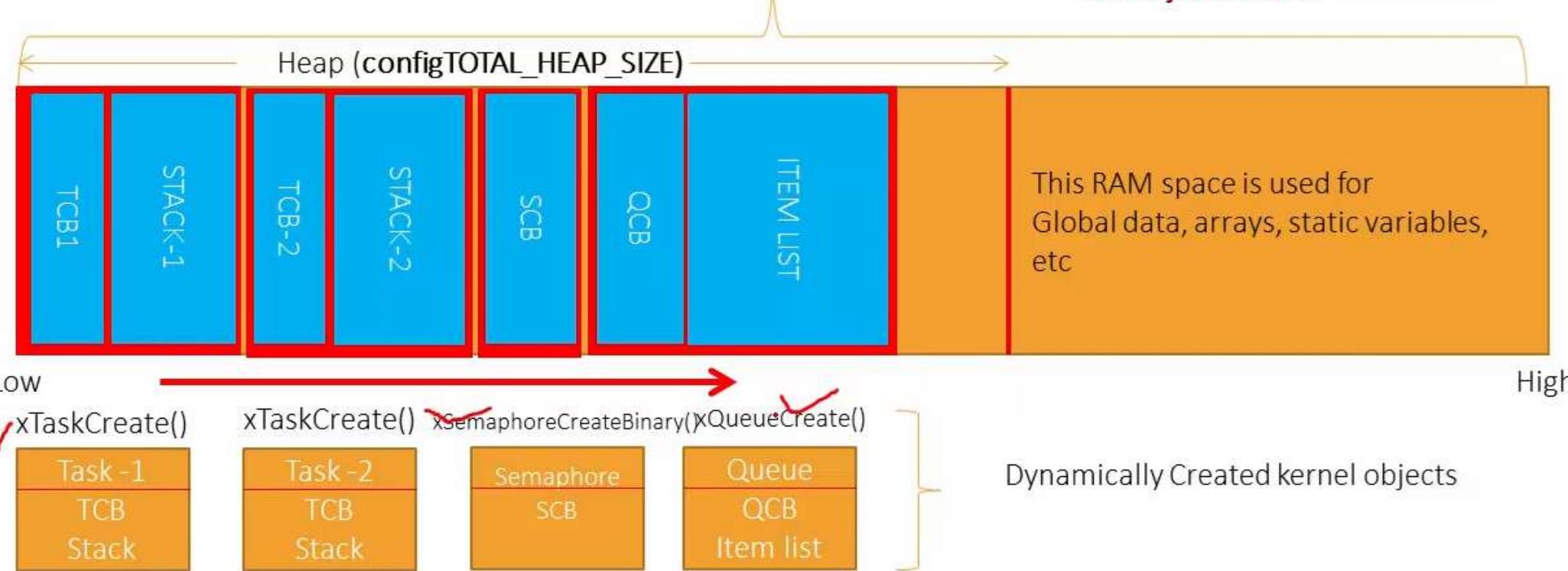


FreeRTOS Stack and heap



FreeRTOS Stack and heap

FreeRTOS Create APIs will internally call memory allocation functions to reserve space in the heap memory of the RAM.



FreeRTOS Heap management Scheme

heap_1.c

heap_2.c

heap_3.c

heap_4.c

heap_5.c

Your_own
_mem.c

pvPortMalloc()

pvPortMalloc()
vPortFree()

pvPortMalloc()
vPortFree()

pvPortMalloc()
vPortFree()

pvPortMalloc()
vPortFree()

pvPortMalloc()
vPortFree()

Application uses any one of these Schemes according to its requirements

FreeRTOS APIs and
Applications

Heap 1.c

Static Memory Allocation

Simplest Implementation among all other heap management implementations

In this implementation you can only allocate Heap memory but you can not free it.

Can be used if your application never deletes a task, queue, semaphore, mutex, etc. (Which actually covers the majority of applications in which Free RTOS gets used).

This implementation is always deterministic (always takes the same amount of time to execute) and cannot result in memory fragmentation.

Heap 2.c

Simple Dynamic Allocation

Supports memory allocation and freeing (`pvPortMalloc()` and `vPortFree()`).

Simple applications that require some dynamic memory allocation.

Allocates and frees memory but does not handle fragmentation well.

Heap 3.c

C Library Malloc/Free

The heap_3.c is not a freeRTOS customization to allocate and de allocate heap, it just a wrapper around the standard 'C' library functions like malloc and free.

Since this implementation uses malloc and free, the code size may shoot up, because malloc and free are not embedded system friendly in terms of code space consumption.

Also using malloc and free will undoubtedly kill the time deterministic nature of the real time application because malloc and free are indeed slow compared to other heap implementations.

Heap 4.c

Best-fit Allocation with Coalescing

Provides allocation and free support with coalescing to avoid fragmentation.

Ideal for applications requiring flexible and dynamic memory allocation with a focus on efficient use of heap space.

Merges adjacent free blocks to reduce fragmentation.

Heap 5.c

Multiple Regions Support

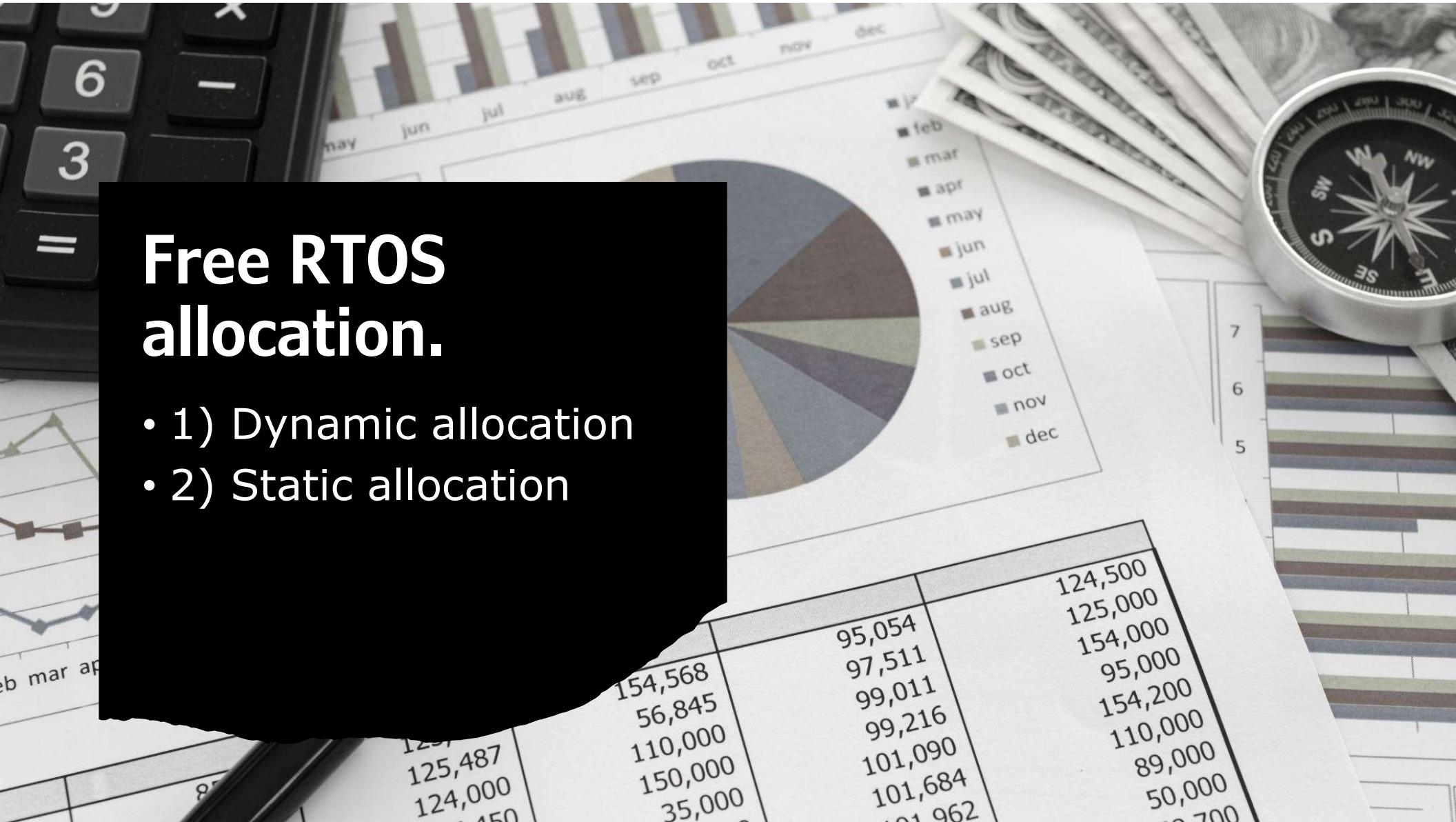
Allows memory allocation across multiple regions of memory.

Useful when your system has non-contiguous blocks of memory.

You can define several memory regions, and FreeRTOS will allocate memory across these regions.

FreeRTOS allocation.

- 1) Dynamic allocation
- 2) Static allocation



Memory Management scheme

The RTOS kernel allocates RAM each time a task, queue, mutex, software timer or semaphore is created.

To replace and improve the use of C standard library malloc() and free() functions, the FreeRTOS download includes four sample memory allocation implementations.
The schemes are implemented in the heap_1.c, heap_2.c, heap_3.c, heap_4.c and heap5.c sources files respectively.

The schemes defined by heap_1.c, heap_2.c, heap_4.c and heap_5.c allocate memory from a statically allocated array, known as the FreeRTOS heap.

TOTAL_HEAP_SIZE sets the size of this array. The size is specified in bytes.

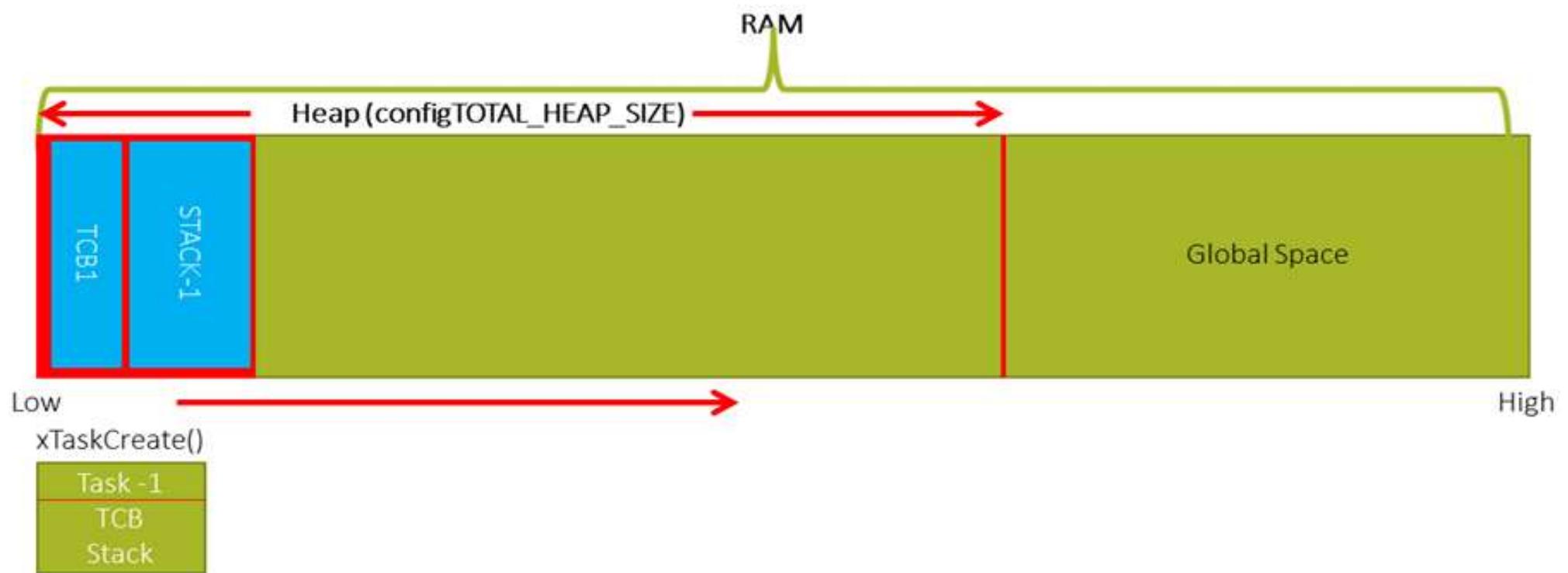
Memory Management scheme

Both allocations required with current CMSIS-RTOS V2 files

In CubeMx, setting MEMORY_ALLOCATION allows to configure configSUPPORT_STATIC_ALLOCATION and configSUPPORT_DYNAMIC_ALLOCATION parameters at the same time:

-
- when equals to "Dynamic", sets configSUPPORT_DYNAMIC_ALLOCATION to 1 and configSUPPORT_STATIC_ALLOCATION to 0
 - when equals to "Static", sets configSUPPORT_DYNAMIC_ALLOCATION to 0 and configSUPPORT_STATIC_ALLOCATION to 1
 - when equals to "Dynamic / Static", sets both configSUPPORT_DYNAMIC_ALLOCATION and configSUPPORT_STATIC_ALLOCATION

Dynamic Memory Allocation From Heap



Static Memory Allocation From Global Space (outside of heap)

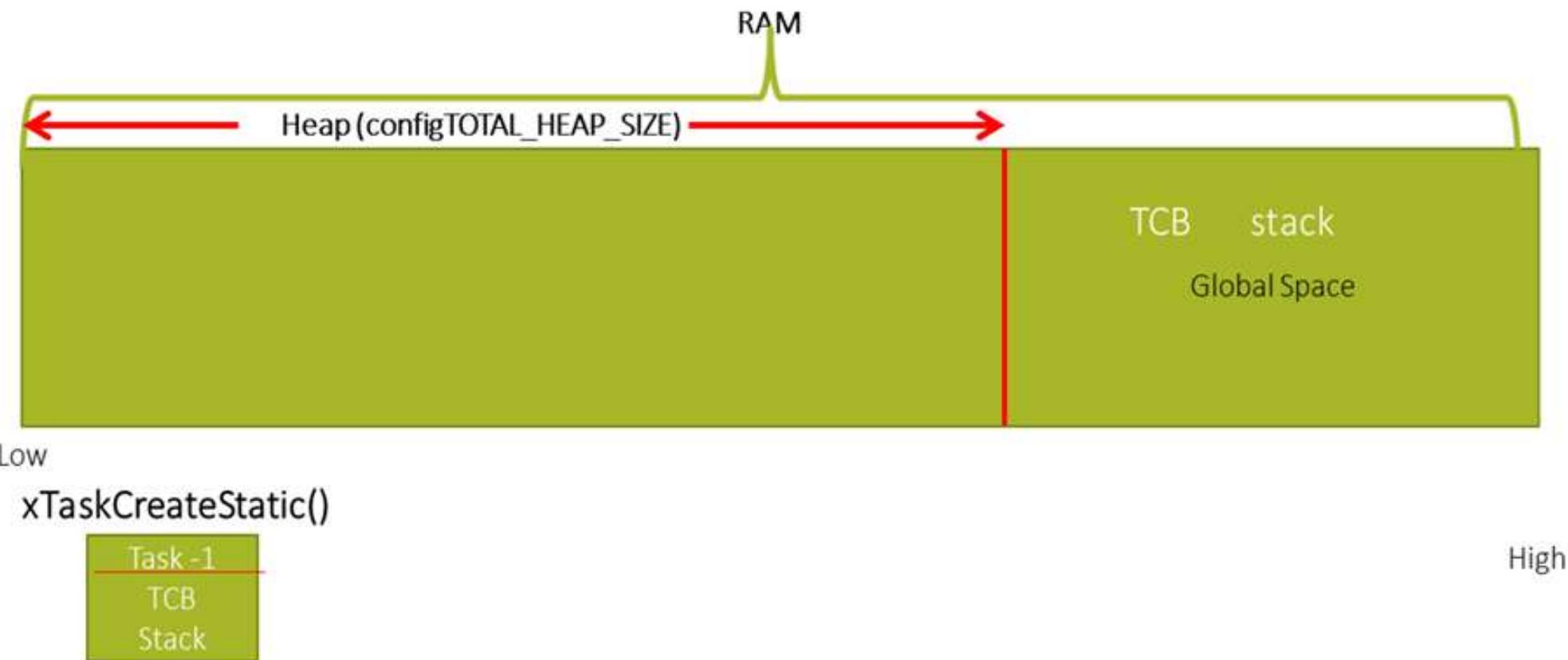
How do you take some part of the RAM? Use Array!!!!

So, the below array is the stack for your task.

```
/* Buffer that the task being created will use as its stack. Note this is  
an array of StackType_t variables. The size of StackType_t is dependent on  
the RTOS port. */  
StackType_t xStack[ STACK_SIZE ];
```

Great! Now you defined both TCB and STACK in the global space of the RAM which looks like this.

Static Memory Allocation From Global Space (outside of heap)



ARM Cortex M Architecture Specific code mainly consists of

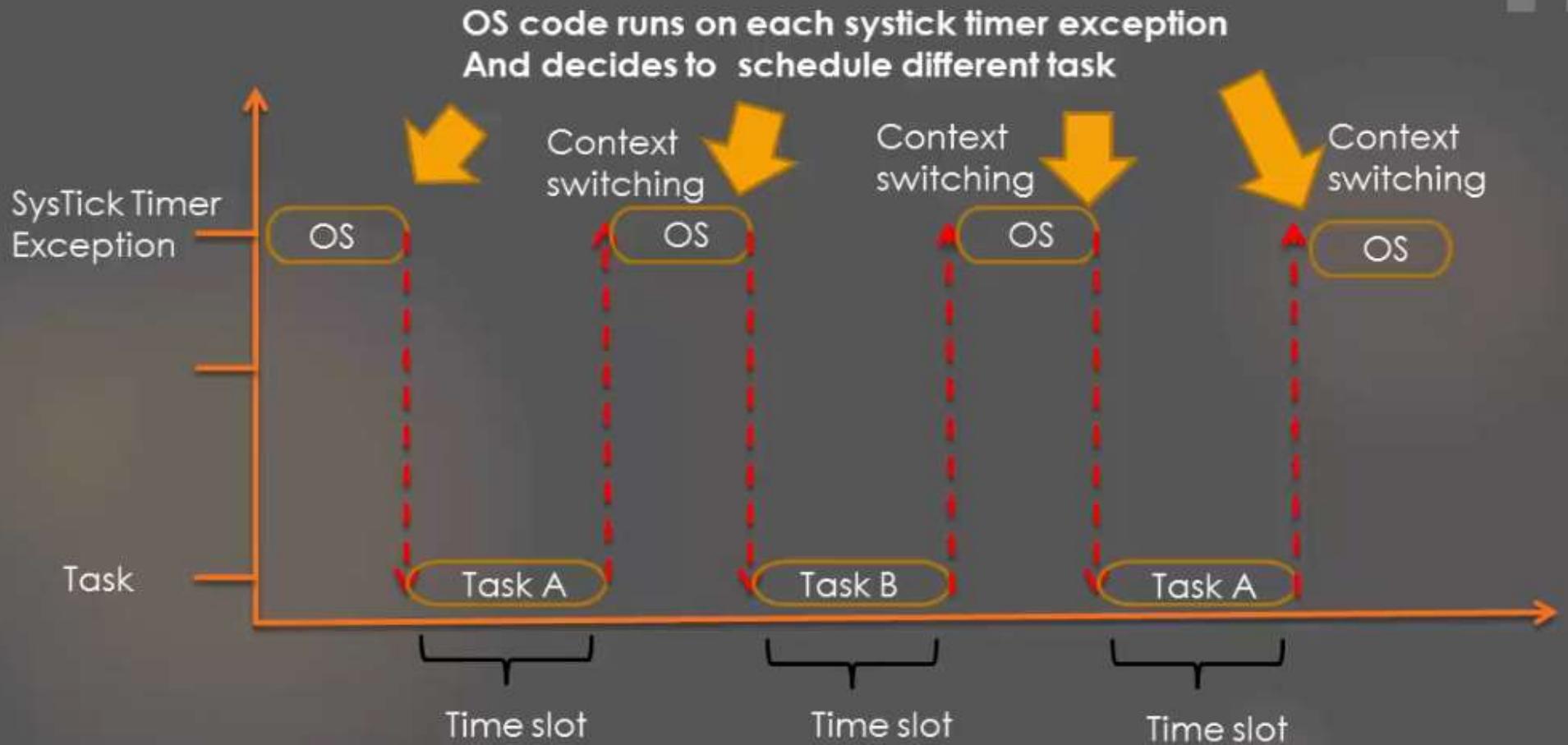
1. SVC Handler (for Scheduling first task)
2. PendSV Handler (for Context Switching)
3. SysTick Handler (for OS ticking)

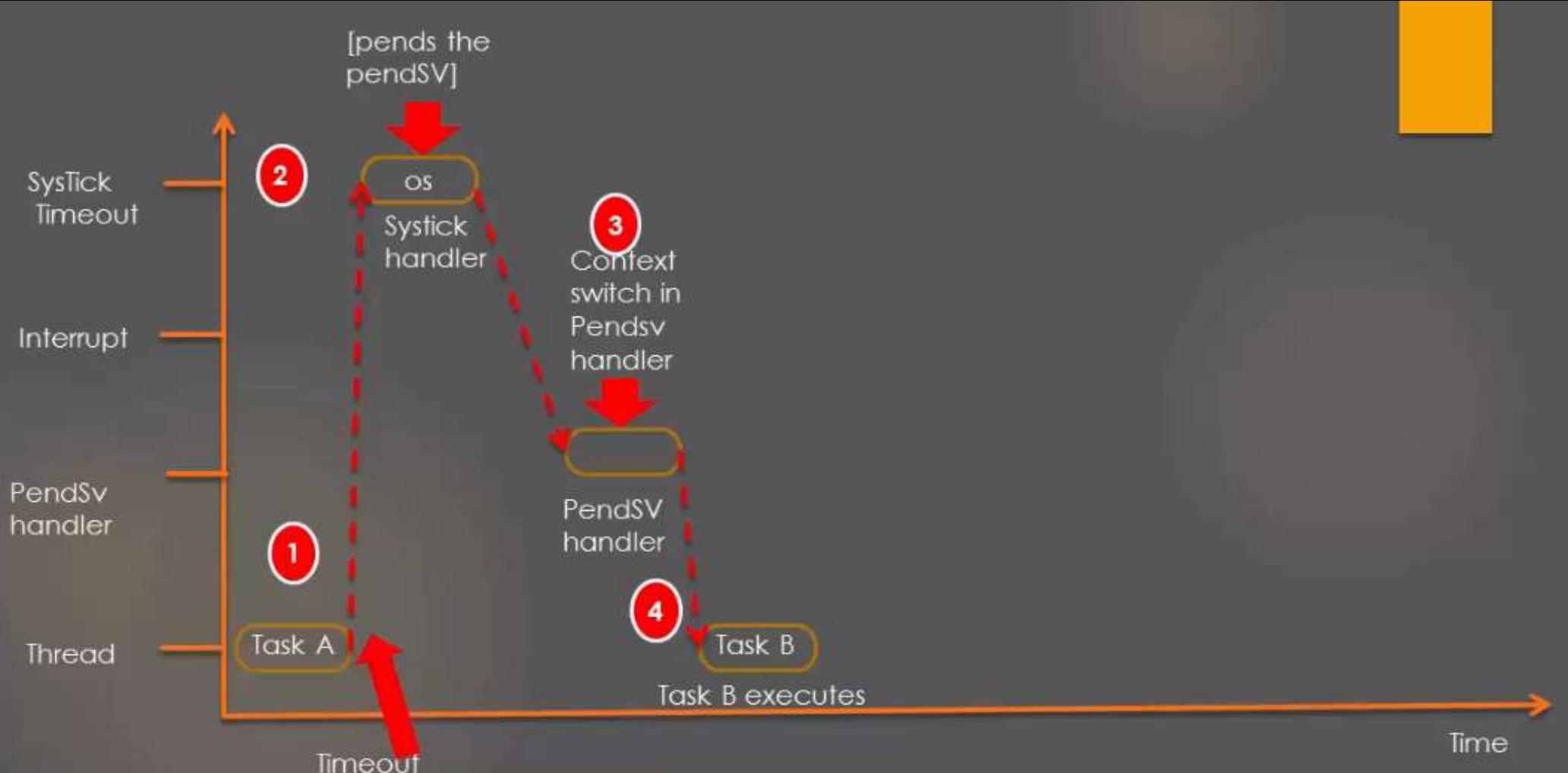
Pend SV Handler

PendSV Exception

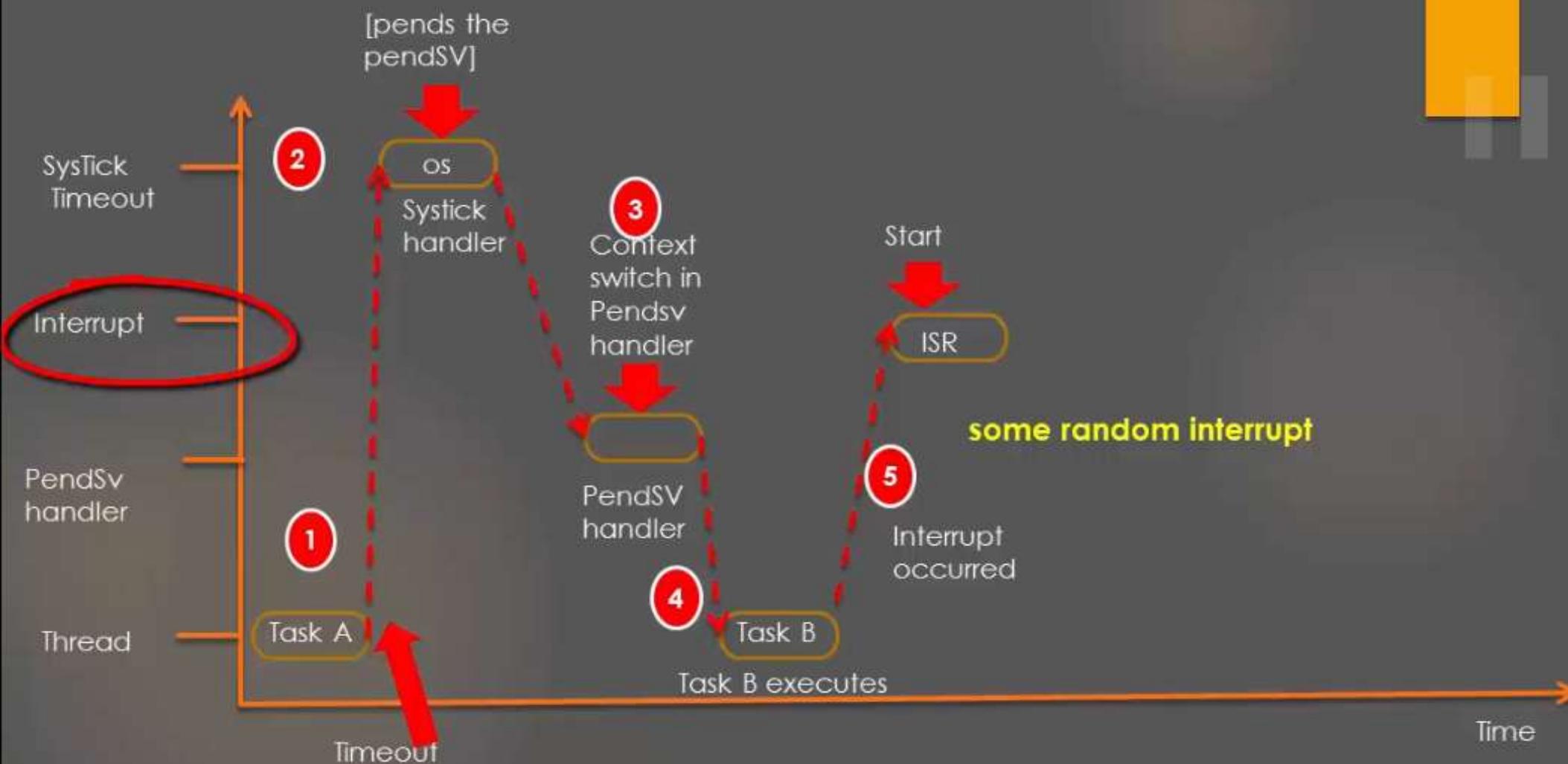
- In OS Designs, we need to switch between different tasks to support multitasking. This is typically called context switching.
- Context switching is usually carried out inside the PendSV exception handler.
- It is exception type 14, mean exception number and has a programmable priority level.
- possible can set to lowest priority.
- This exception is triggered by setting its pending status by writing to the "Interrupt Control and State Register".

Context Switching

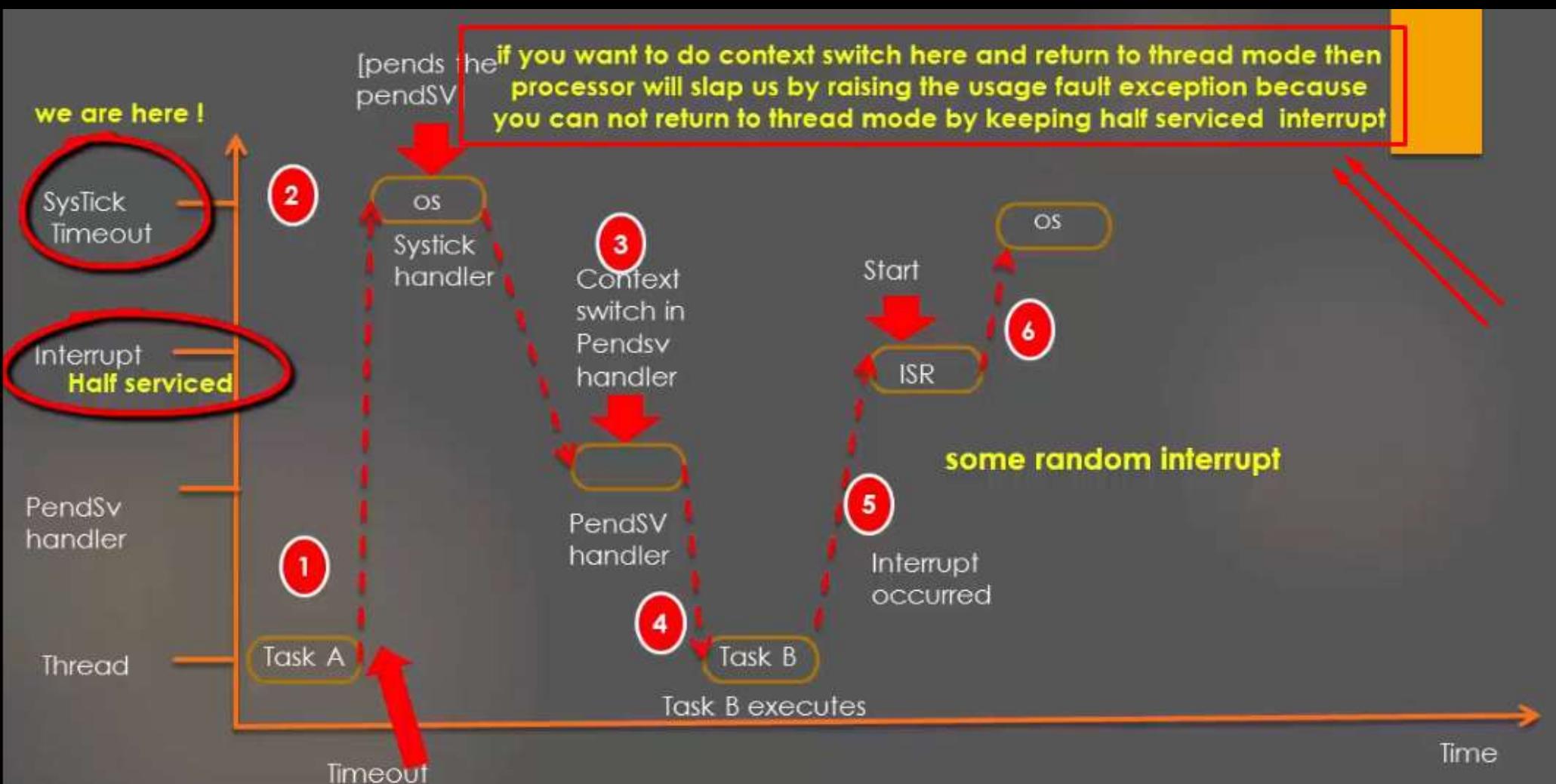




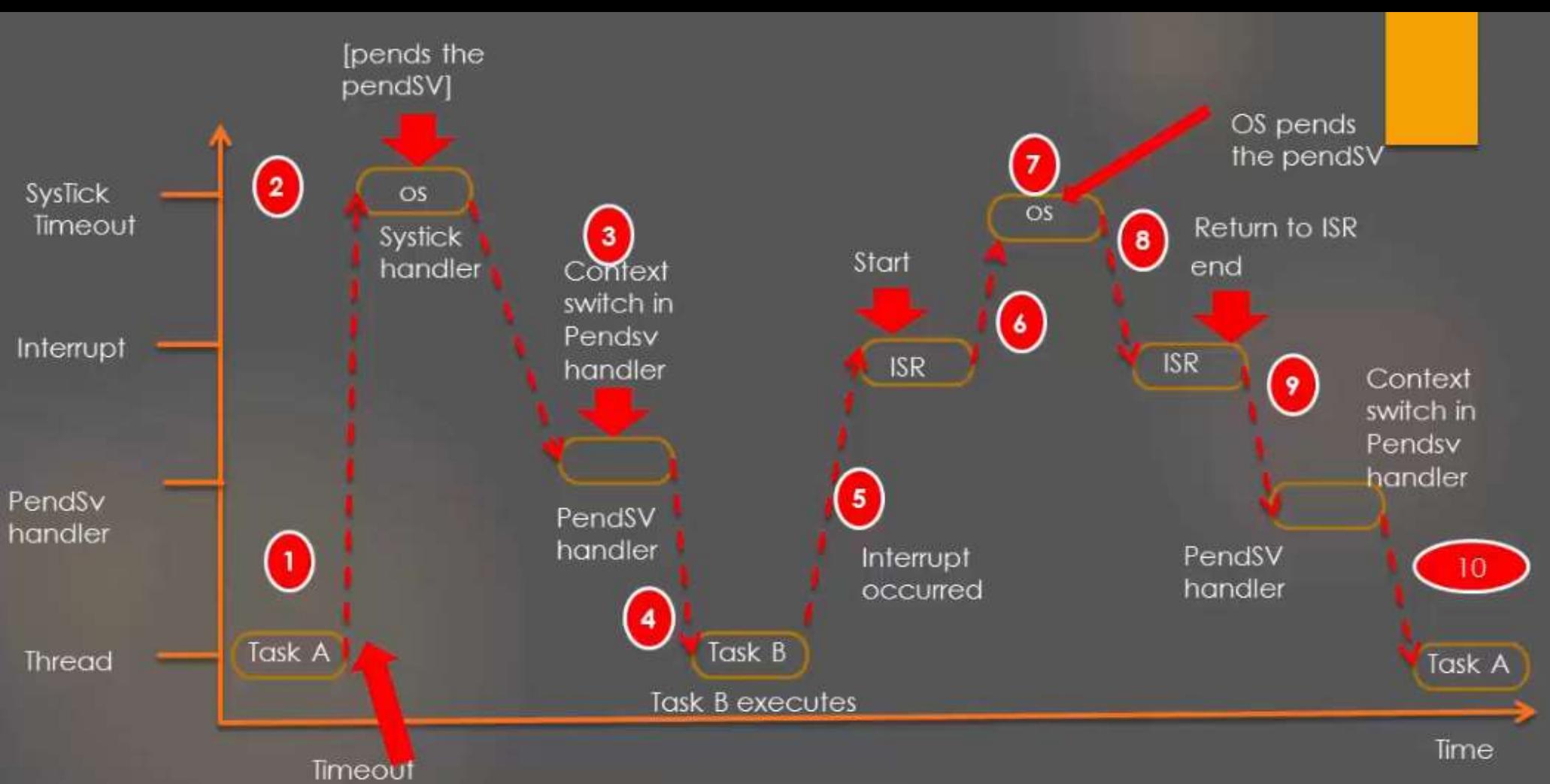
Scenario of PendSV in context switching



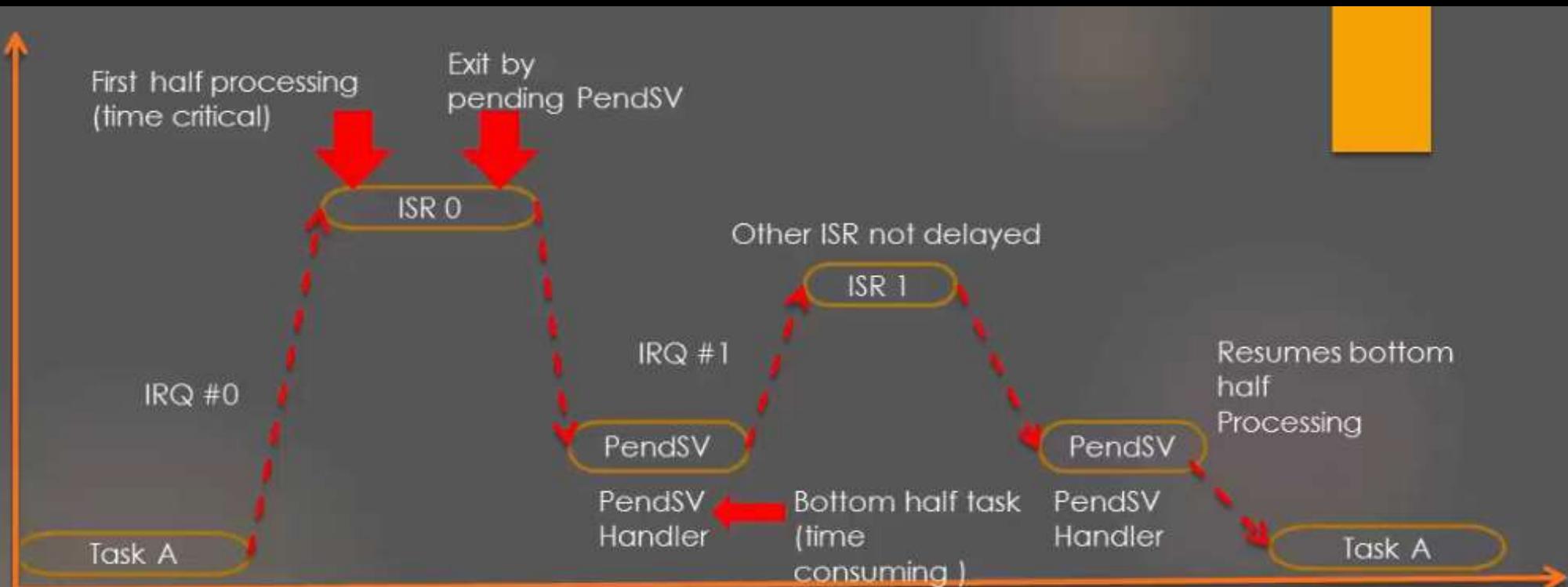
Scenario of PendSV in context switching



Scenario of PendSV in context switching

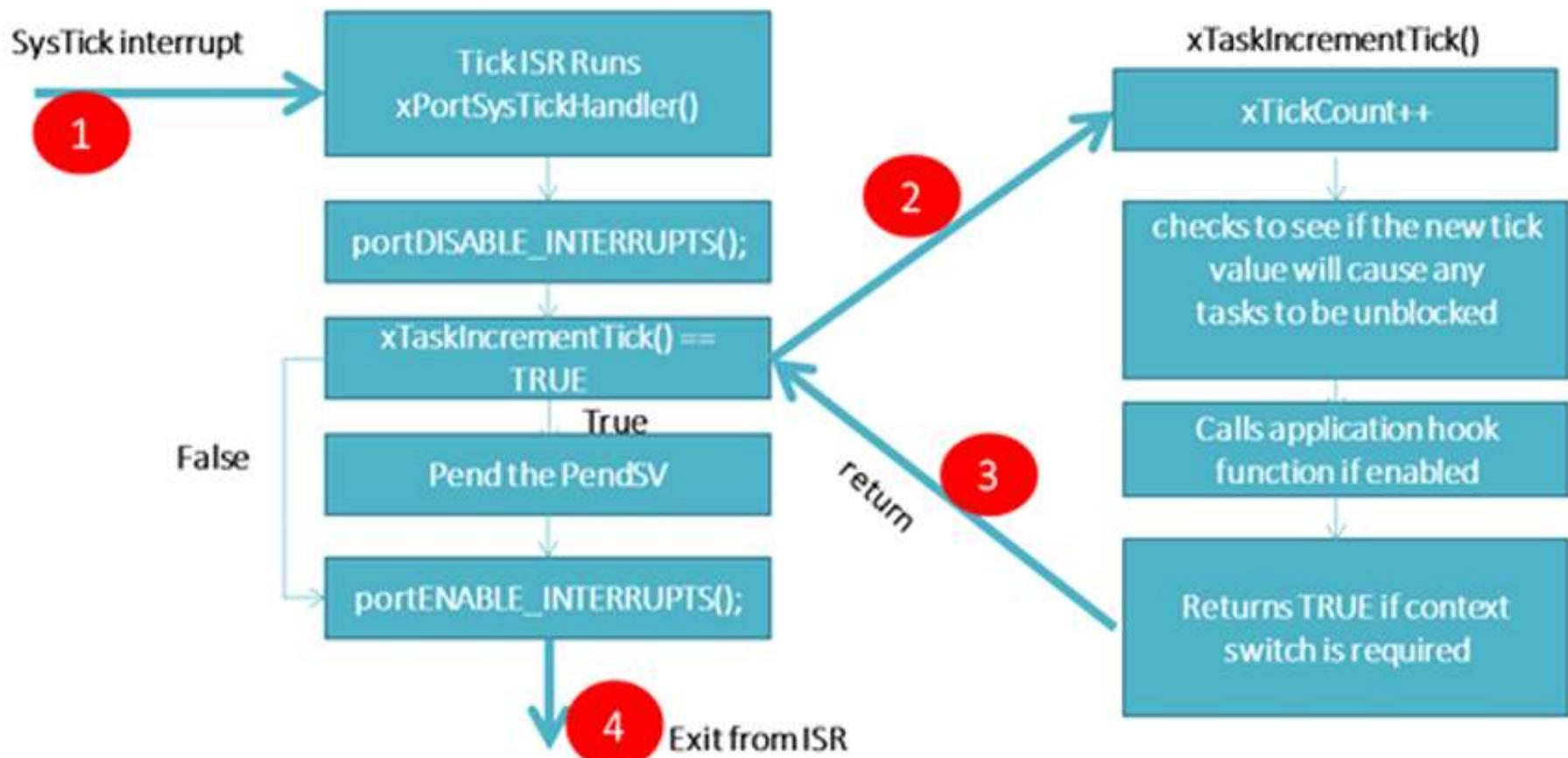


Scenario of PendSV in context switching



Scenario of using PendSV in offloading interrupt processing

What RTOS tick ISR does ? : Conclusion



configTICK_RATE_HZ

The frequency of the RTOS tick interrupt.

**This decides the frequency of the RTOS timer interrupt !
if it is 1000Hz , then RTOS timer interrupts the processor for every
1ms.
if its value is 250Hz , then RTOS timer interrupts the processor for
every 4ms.**

The tick interrupt is used to measure time. Therefore a higher tick frequency means time can be measured to a higher resolution. However, a high tick frequency also means that the RTOS kernel will use more CPU time so be less efficient. The RTOS demo applications all use a tick rate of 1000Hz. This is used to test the RTOS kernel and is higher than would normally be required.

More than one task can share the same priority. The RTOS scheduler will share processor time between tasks of the same priority by switching between the tasks during each RTOS tick. A high tick rate frequency will therefore also have the effect of reducing the 'time slice' given to each task.

CPU_CLK_HZ

```
#define configCPU_CLOCK_HZ      (sysclk_get_cpu_hz() )
```

The macro configCPU_CLOCK_HZ uses the “sysclk_get_cpu_hz” function from ASF to retrieve the frequency of the CPU.



In order to ensure that sysclk_get_cpu_hz() return the correct system frequency, the “sysclk_init()” function should be added at the beginning of the main routine, to ensure that correct frequency configuration is applied during code execution.

`configTICK_RATE_HZ`

The FreeRTOS uses a timer peripheral which we call as OS timer and it generates interrupts and this Macro defines the rate or frequency of those interrupts.

The frequency of the RTOS tick interrupt

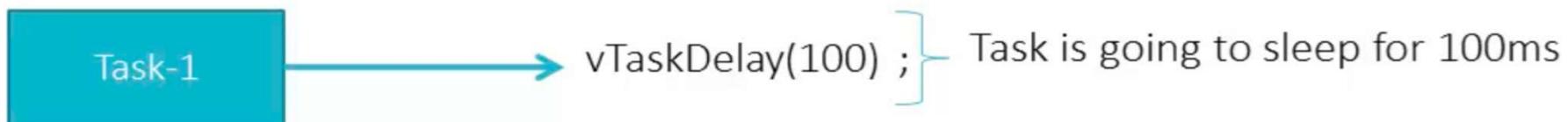


The tick interrupt is used to measure time. Therefore a higher tick frequency means time can be measured to a higher resolution. However, a high tick frequency also means that the RTOS kernel will use more CPU time so be less efficient. The RTOS demo applications all use a tick rate of 1000Hz. This is used to test the RTOS kernel and is higher than would normally be required.

More than one task can share the same priority. The RTOS scheduler will share processor time between tasks of the same priority by switching between the tasks during each RTOS tick. A high tick rate frequency will therefore also have the effect of reducing the 'time slice' given to each task.

- Tick interrupt happen every 1 ms in 1000 Hz
- Tick interrupt happen every 4 ms in 250 Hz

The RTOS Tick- Why it is needed ?



Now, its a duty of the Kernel to wake up the task after 100ms.

So, How do you think the freeRTOS kernel tracks the completion of 100ms ?

FreeRTOS calculates the time, with reference to "tick_count" variable which increments on every OS timer interrupt.

If configTICK_RATE_HZ=1000Hz, and if "tick_count" variable is incremented by 100, then we can say, 100ms have been elapsed.

So, this task will be woken up by checking the "tick_count" variable for every tick interrupt! If, it is incremented by 100, then task will be woken up.

The RTOS Tick- Why it is needed ?

Used for Context Switching to the
next potential Task

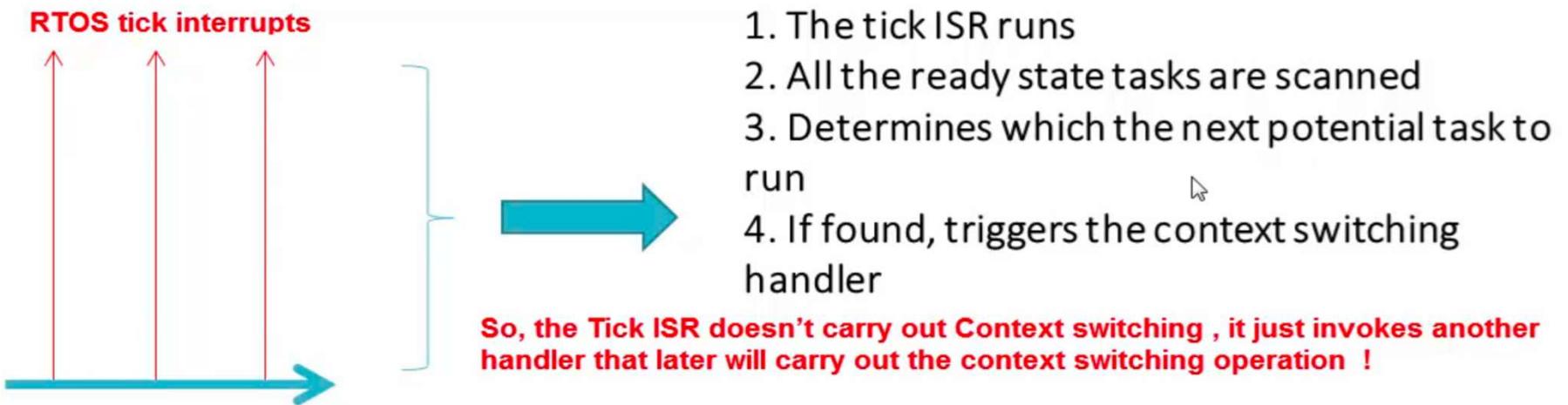
in a multitaskable RTOS environment in a multitasking RTOS environment context switching is a primary need and OS tick interrupts will help to implement that.

Swapping out the context of the running task and retrieving the context of the next potential task to run on the CPU is called Context Switching.



The RTOS Tick- Why it is needed ?

Used for Context Switching to the next potential Task



- Kernel will be use hardware timer peripherals to issues ticks

```
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
port.c X

435
436 void xPortSysTickHandler( void )
437 {
438     /* The SysTick runs at the lowest interrupt priority, so when this interrupt
439     executes all interrupts must be unmasked. There is therefore no need to
440     save and then restore the interrupt mask value as its value is already
441     known. */
442     portDISABLE_INTERRUPTS();
443     {
444         /* Increment the RTOS tick */
445         if( xTaskIncrementTick() != pdFALSE )
446             {
447                 /* A context switch is required. Context switching is performed in
448                 the PendSV interrupt. Pend the PendSV interrupt. */
449                 portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;
450             }
451     }
452     portENABLE_INTERRUPTS();
453 }
454 /*-----*/
455
456 #if configUSE_TICKLESS_IDLE == 1
457     __attribute__((weak)) void vPortSuppressTicksAndSleep( TickType_t xExpectedIdleTime )
source file length : 28172 lines : 711 Ln : 445 Col : 31 Sel : 18 | 0 Dos\Windows UTF-8 INS

```

As i said previously, whenever tick interrupt happens, the "tick_count" variable will be incremented. That is done in this function!

This function also scans all the ready state tasks to select the next potential task to run on the cpu.

Task Implementation Function(Task Function)

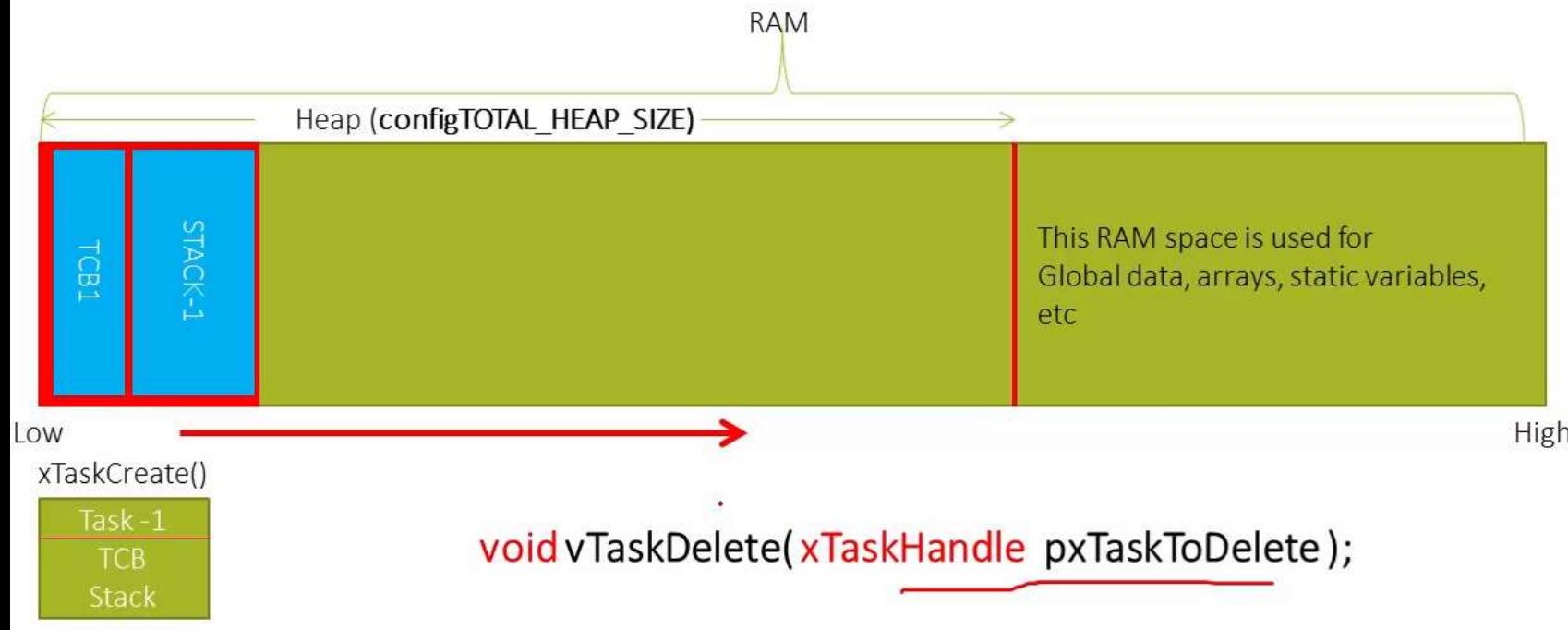
```
void ATaskFunction( void *pvParameters )
{
    /* Variables can be declared just as per a normal function. Each instance
       of a task created using this function will have its own copy of the
       iVariableExample variable. This would not be true if the variable was
       declared static - in which case only one copy of the variable would exist
       and this copy would be shared by each created instance of the task. */
    int iVariableExample = 0; -----^

    /* A task will normally be implemented as in infinite loop. */
    for( ;; )
    {
        /* The code to implement the task functionality will go here. */
    }

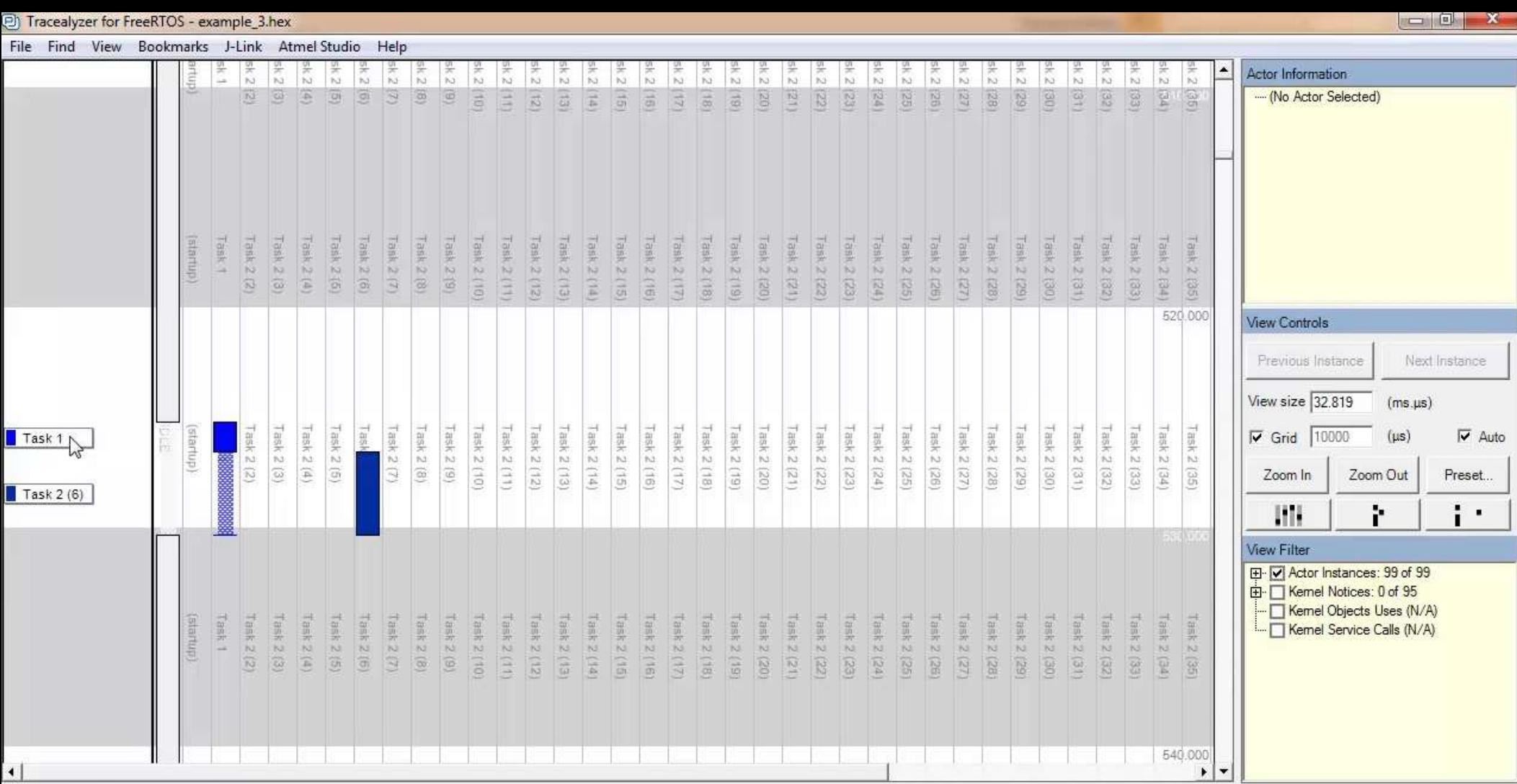
    /* Should the task implementation ever break out of the above loop
       then the task must be deleted before reaching the end of this function.
       The NULL parameter passed to the vTaskDelete() function indicates that
       the task to be deleted is the calling (this) task. */
    vTaskDelete( NULL );
}
```

- vTaskDelete() ወ task ተወቃዕሚ function ጽሑፍ አለው፡ይህንና፡ ማረጋገጫ፡
- የሁላ - task2 ወጪ ይመዘግል፡ function handler የውጭ፡ማይሸጥበት int ቁጥር፡ iVariableExample ዘመን፡ task ተወቃዕሚ፡ወጪ፡ local var ዘመን፡ Task ስም፡ stack memory መካከል፡ሸጥበት፡
- Int የውጭ፡ static ዘመን፡ global shared variable ዘመን፡ task ተወቃዕሚ፡ task ዘመን፡

Deleting a Task

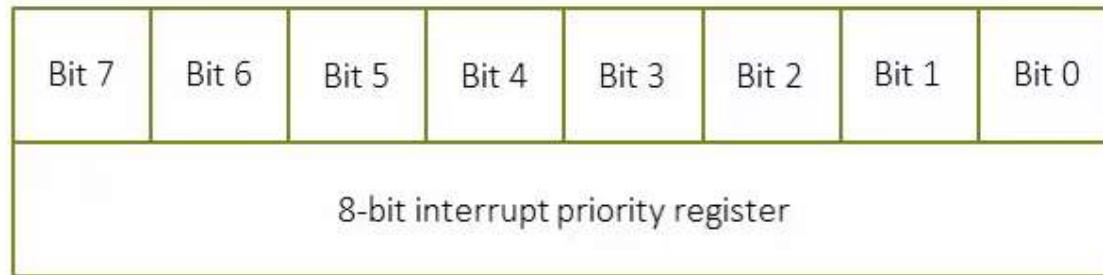


- Task ကို delete လုပ်တော့ memory ပေါ်က TCB နဲ့ Stack memory တို့ကို deallocate လုပ်ပေးခြင်းမဟုတ်ပါ။
- Task ကို delete လုပ်ထားကြောင်း declare လုပ်ပေးရုံသာဖြစ်တယ်။
- Rtos system ၉၀ % သုံးသော task များသည် delete လုပ်ရန်မလိုအပ်ပါ။
- ထို့ကြောင့် task တစ်ခုကို မ run စေချင်ရင် blocked state (or) suspended state ဘားရန်အကြံပေး။



Priority register and priority levels in ARM Cortex-M Processor

Priority register in Cortex M3/M4



In Cortex M based Processor every interrupt and System exception has 8 bit interrupt priority register to configure its Priority

So , ideally there are 2^8 (256) interrupt priority levels from 0x00 to 0xff.

Where 0x00 is the highest priority and 0xff is the lowest priority

Priority Register

Microcontroller Vendor XXX

TM4C123G

✓ *m4*

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Implemented	Not implemented						



8 Levels of Priority level



✓ 0x00,0x20,0x40,0x60,
0x80,0xA0,0xC0,0xE0
✓

Microcontroller Vendor YYY

✓ STM32F4xx

AT91SAM3X8E

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Implemented	Not implemented						



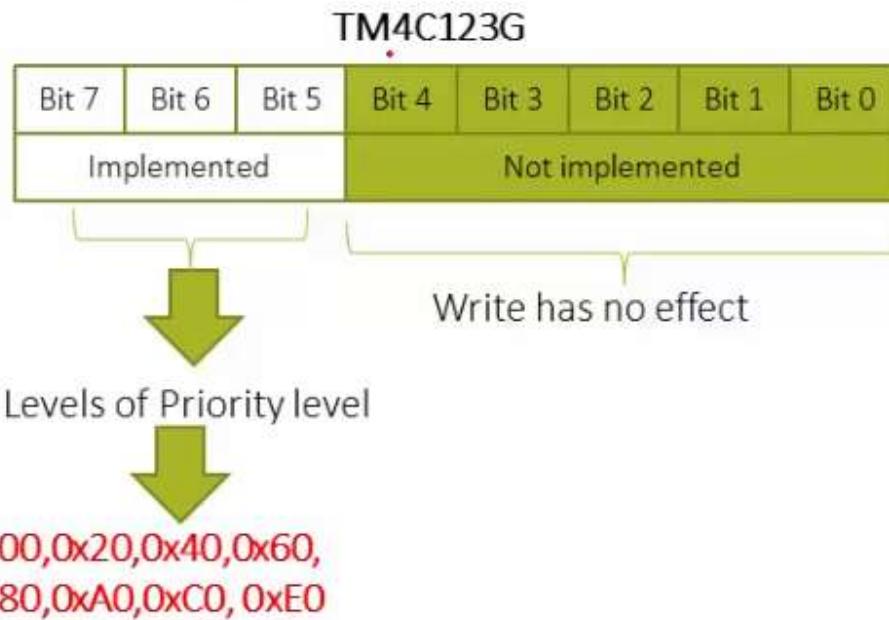
16 Levels of Priority level



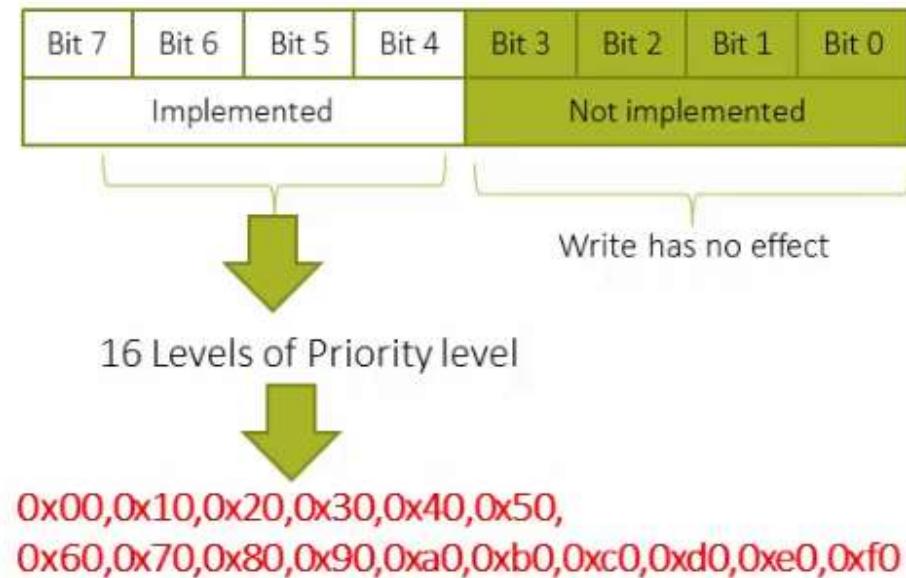
0x00,0x10,0x20,0x30,0x40,0x50,
0x60,0x70,0x80,0x90,0xa0,0xb0,0xc0,0xd0,0xe0,0xf0

Even though Priority register is 8bit, the vendor who fabricates the MCU may not implement all the bits of the Priority register. let's say if ATTEL wants to implement only 4 bits, then they just want to give only 16 different priority levels. That means at worst case 16 different interrupts can nest. If they allow all 256 levels then it may blow up the RAM usage at the worst case and makes system complex in terms of interrupt handling. So most of the time you will see either 3 or 4 bits are implemented.

Microcontroller Vendor XXX



Microcontroller Vendor YYY



Example 1 : Setting Priority

Let's say you want to configure a priority of an interrupt number 8(IRQ8) to be 5.

AT91SAM3X8E							
Interrupt priority Register corresponding to IRQ8							
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Implemented					Not implemented		
0	0	0	0	0	1	0	1

s = 5

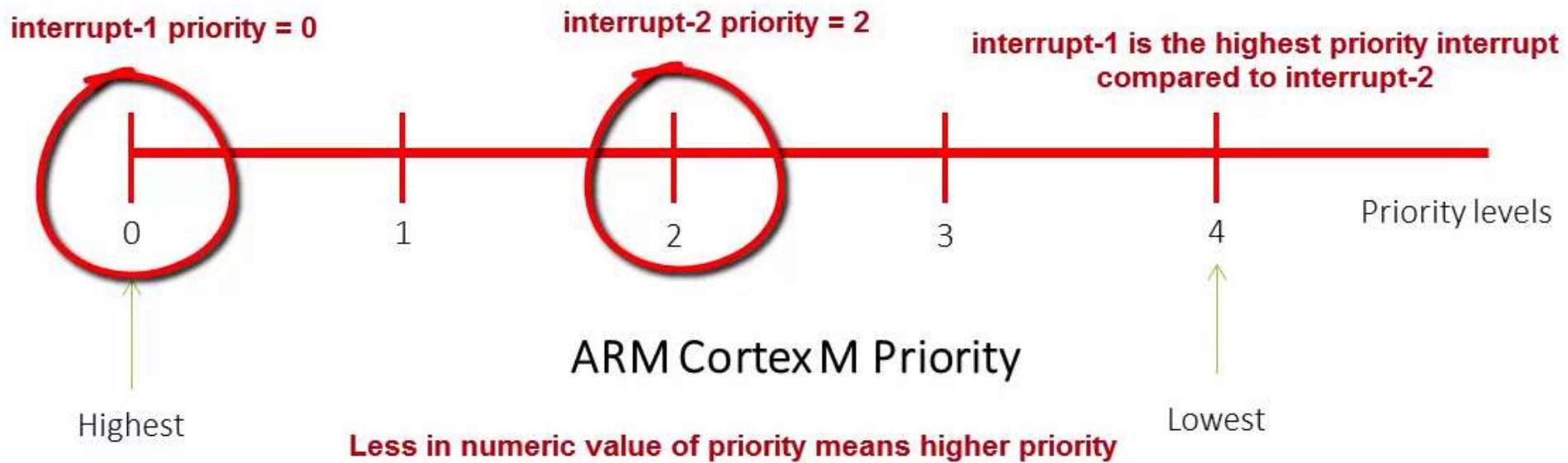
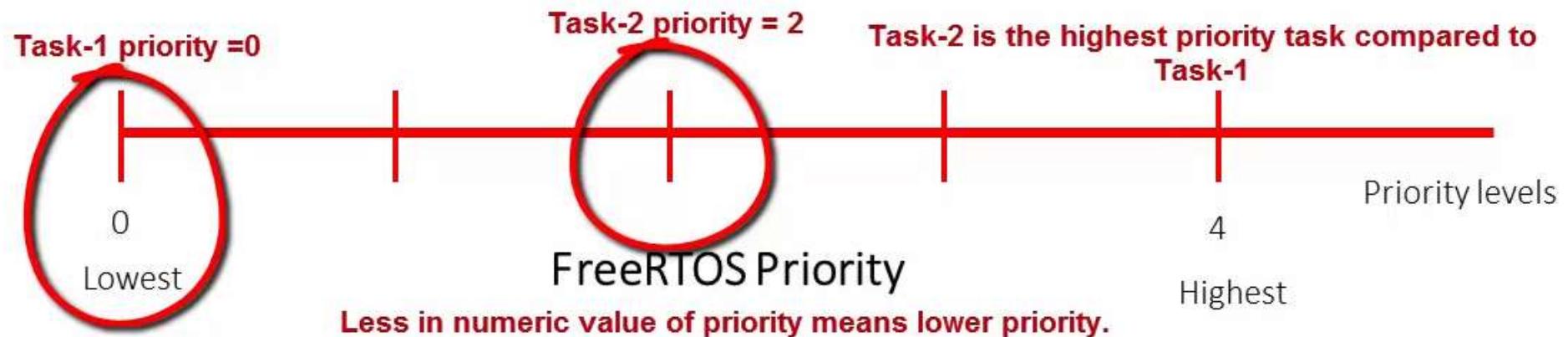
Priority_register = priority_value

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Implemented					Not implemented		
0	1	0	1	0	0	0	0

Priority_register = priority_value << (8 - __NVIC_PRIO_BITS)

$$5 \ll (8 - 4)$$

FreeRTOS Task Priority Vs Processor Interrupt/Exception Priority



FreeRTOS Priority Configuration Config Items

configKERNEL_INTERRUPT_PRIORITY

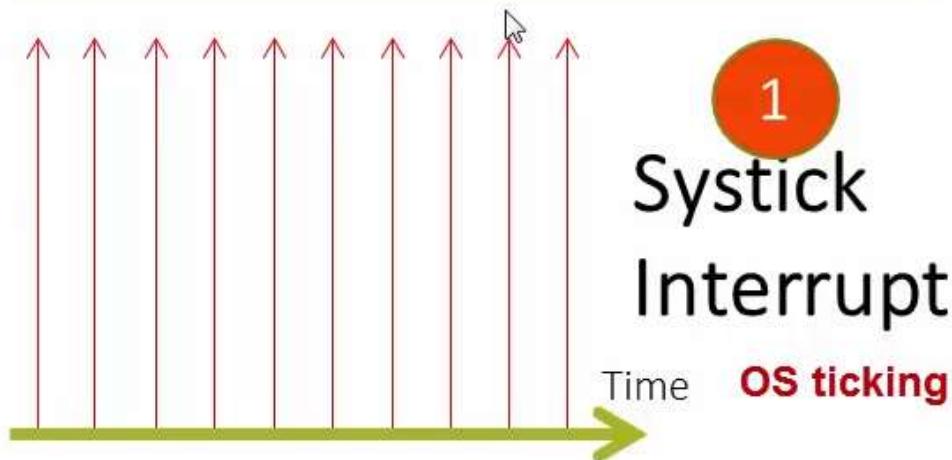
configMAX_SYSCALL_INTERRUPT_PRIORITY

configKERNEL_INTERRUPT_PRIORITY

This config item, decides the priority for the kernel Interrupts

What are the kernel interrupts ?

```
#define configTICK_RATE_HZ ( (portTickType)1000)
```



```
MX CMSIS_RTOS_F4.ioc    c main.c    h FreeRTOSConfig.h X

134 /* The lowest interrupt priority that can be used in a call to a "set priority"
135 function. */
136 #define configLIBRARY_LOWEST_INTERRUPT_PRIORITY 15
137
138 /* The highest interrupt priority that can be used by any interrupt service
139 routine that makes calls to interrupt safe FreeRTOS API functions. DO NOT CALL
140 INTERRUPT SAFE FREERTOS API FUNCTIONS FROM ANY INTERRUPT THAT HAS A HIGHER
141 PRIORITY THAN THIS! (higher priorities are lower numeric values. */
142 #define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY 5
143
144 /* Interrupt priorities used by the kernel port layer itself. These are generic
145 to all Cortex-M ports, and do not rely on any particular library functions. */
146 #define configKERNEL_INTERRUPT_PRIORITY ( configLIBRARY_LOWEST_INTERRUPT_PRIORITY << (8 - configPrio_Bits) )
147 /* !!!! configMAX_SYSCALL_INTERRUPT_PRIORITY must not be set to zero !!!!
148 See http://www.FreeRTOS.org/RTOS-Cortex-M3-M4.html. */
149 #define configMAX_SYSCALL_INTERRUPT_PRIORITY ( configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY << (8 - configPrio_Bits) )
150
151
152 header file. */
153 /* USER CODE BEGIN 1 */
154 #define configASSERT( x ) if ((x) == 0) {taskDISABLE_INTERRUPTS(); for( ; ; );}
155 /* USER CODE END 1 */
156
157 /* Definitions that map the FreeRTOS port interrupt handlers to their CMSIS
158 standard names. */
159 #define vPortSVCHandler SVC_Handler
160 #define xPortPendSVHandler PendSV_Handler
161
162 /* IMPORTANT: After 10.3.1 update, Systick_Handler comes from NVIC (if SYS_timebase = systick), otherwise from cmsis_os2.c */
163
164 #define USE_CUSTOM_SYSTICK_HANDLER_IMPLEMENTATION 0
165
166 /* USER CODE BEGIN Defines */
167 /* Section where parameter definitions can be added (for instance, to override default ones in FreeRTOS.h) */
168 /* USER CODE END Defines */
```

```
/*
 * \brief Configuration of the Cortex-M3 Processor and Core Peripherals
 */

#define __CM3_REV          0x0200 /*< SAM3A4C core revision number ([15:8] revision number, [7:0] patch number) */
#define __MPU_PRESENT       1      /*< SAM3A4C does provide a MPU */
#define __NVIC_PRIO_BITS    4      /*< SAM3A4C uses 4 Bits for the Priority Levels */
#define __Vendor_SysTickConfig 0  /*< Set to 1 if different SysTick Config is used */

/*
```

FreeRTOSConfig.h X sam3a4c.h X

```
function. */
#define configLIBRARY_LOWEST_INTERRUPT_PRIORITY      0x0F

/* The highest interrupt priority that can be used by any interrupt service
routine that makes calls to interrupt safe FreeRTOS API functions. DO NOT CALL
INTERRUPT SAFE FREERTOS API FUNCTIONS FROM ANY INTERRUPT THAT HAS A HIGHER
PRIORITY THAN THIS! (higher priorities are lower numeric values. *)
#define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY 10

/* Interrupt priorities used by the kernel port layer itself. These are generic
to all Cortex-M ports, and do not rely on any particular library functions. */
#define configKERNEL_INTERRUPT_PRIORITY      ( configLIBRARY_LOWEST_INTERRUPT_PRIORITY << (8 - configPRIO_BITS) )
#define configMAX_SYSCALL_INTERRUPT_PRIORITY     ( configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY << (8 - configPRIO_BITS) )

/* Normal assert() semantics without relying on the provision of an assert.h
   file. */
```

<input checked="" type="checkbox"/> Mutexes	<input checked="" type="checkbox"/> Events	<input checked="" type="checkbox"/> FreeRTOS Heap Usage
<input checked="" type="checkbox"/> User Constants	<input checked="" type="checkbox"/> Tasks and Queues	<input checked="" type="checkbox"/> Timers and Semaphores
<input checked="" type="checkbox"/> Config parameters	<input checked="" type="checkbox"/> Include parameters	<input checked="" type="checkbox"/> Advanced settings

Configure the below parameters :

 Search (Ctrl+F)



▼ Interrupt nesting behaviour configuration

LIBRARY_LOWEST_INTERRUPT_PRIORITY 15

LIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY 5

▼ Added with 10.2.1 support

MESSAGE_BUFFER_LENGTH_TYPE size_t

USE_POSIX_ERRNO Disabled

LIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY

LIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY must be between 1 and 15.

Diagnostic:

Available for series with a Cortex core that is not a M0.

Parameter Description:

The highest interrupt priority that can be used by any interrupt service routine that makes calls to interrupt safe FreeRTOS API functions.

DO NOT CALL INTERRUPT SAFE FREERTOS API FUNCTIONS FROM ANY
INTERRUPT THAT HAS A HIGHER PRIORITY THAN THIS! (higher priorities are lower
numeric values).

Concluding Points

First we learnt there are 2 configuration items

`configKERNEL_INTERRUPT_PRIORITY`

`configMAX_SYSCALL_INTERRUPT_PRIORITY`

`configKERNEL_INTERRUPT_PRIORITY`: The kernel interrupt priority config item actually decides the priority level for the kernel related interrupts like systick, pendsv and svc and it is set to lowest interrupt priority as possible.

`configMAX_SYSCALL_INTERRUPT_PRIORITY`: The max sys call interrupt priority config item actually decides the maximum priority level , that is allowed to use for those interrupts which use freertos APIs ending with “FromIsr” in their interrupt service routines.

<input checked="" type="checkbox"/> Mutexes	<input checked="" type="checkbox"/> Events	<input checked="" type="checkbox"/> FreeRTOS Heap Usage
<input checked="" type="checkbox"/> User Constants	<input checked="" type="checkbox"/> Tasks and Queues	<input checked="" type="checkbox"/> Timers and Semaphores
<input checked="" type="checkbox"/> Config parameters	<input checked="" type="checkbox"/> Include parameters	<input checked="" type="checkbox"/> Advanced settings

Configure the below parameters :

Search (Ctrl+F)	↶	↷	i
CPU_CLOCK_HZ			SystemCoreClock
TICK_RATE_HZ			1000
MAX_PRIORITIES			56
MINIMAL STACK SIZE			128 Words

MAX_PRIORITIES

MAX_PRIORITIES

Diagnostic:

Default set to 56 when CMSIS RTOS V2 is Enabled

Parameter Description:

Sets the maximum priority that can be assigned to a task.

Tasks can be assigned a priority from zero, which is the lowest priority, to (MAX_PRIORITIES-1), which is the highest priority. **Warning:**

Set to 56 when CMSIS-RTOS v2 is in use.

Interrupt Un-Safe APIs

FreeRTOS APIs which don't end with the word “**FromISR**” are called as interrupt unsafe APIs

e.g.

xTaskCreate(),

xQueueSend()

xQueueReceive()

etc

**These APIs should not
be called from an ISR**

**These APIs can only be called from the Task
Context, never use them in the interrupt
context**

Interrupt Safe and Un-safe APIs

If you want to send a data item in to the queue from the ISR, then use `xQueueSendFromISR()` instead of `xQueueSend()`.

`xQueueSendFromISR()` is an interrupt safe version of `xQueueSend()`.

If you want to Read a data item from the queue being in the ISR, then use `xQueueReceiveFromISR()` instead of `xQueueReceive()`.

`xQueueReceiveFromISR()` is an interrupt safe version of `xQueueReceive()`.

`xSemaphoreTakeFromISR()` is an interrupt safe version of `xSemaphoreTake()` : which is used to ‘take’ the semaphore

`xSemaphoreGiveFromISR()` is an interrupt safe version of `xSemaphoreGive()` : which is used to ‘give’ the semaphore

handler

```
QUEUE some_queue;
```

```
ISR_Fun(void *data)  
{ . }
```

```
Queue_write(&some_queue, data);
```

```
next statement 1;  
next statement 2;  
. . .
```

```
}
```

Scenario of an ISR calling non-interrupt safe API

In ARM Cortex M processor keeping a handler mode half done and transitioning into the thread mode is not allowed, doing so will raise an exception called "Usage Fault exception"

```
Queue_write(QUEUE *qptr, void * data)  
{  
    1. write to queue  
    2. does write to queue unblock any higher priority task?  
    3. if yes, must do taskYIELD()  
    4. if no, continue with the same task 1  
}
```

Task-2
Thread

```

QUEUE some_queue;

ISR_Fun(void *data)
{
    unsigned long xHigherPriorityTaskWoken = FALSE;
    Queue_write_FromISR(&some_queue, data, &xHigherPriorityTaskWoken);

    next statement 1;
    next statement 2;
    .
    .
    /* yielding to task happens in ISR Context, no tin API context */
    if(xHigherPriorityTaskWoken)
        portYIELD();
}

```

 Queue_write_FromISR (QUEUE *qptr, void * data, void * xHigherPriorityTaskWoken)

{

1. write to queue
2. does writing to queue unblocks any higher priority task?
3. if yes, then set xHigherPriorityTaskWoken = TRUE
4. if no, then set xHigherPriorityTaskWoken = FALSE
5. return to ISR

}

Scenario of an ISR calling interrupt safe API

Interrupt Safe APIs: Conclusion

Whenever you want use FreeRTOS API from an ISR its ISR version must be used, which ends with the word “FromISR”

This is for the reason, Being in the interrupt Context (i.e being in the middle of servicing the ISR) you can not return to Task Context (i.e making a task to run by pre-empting the ISR)

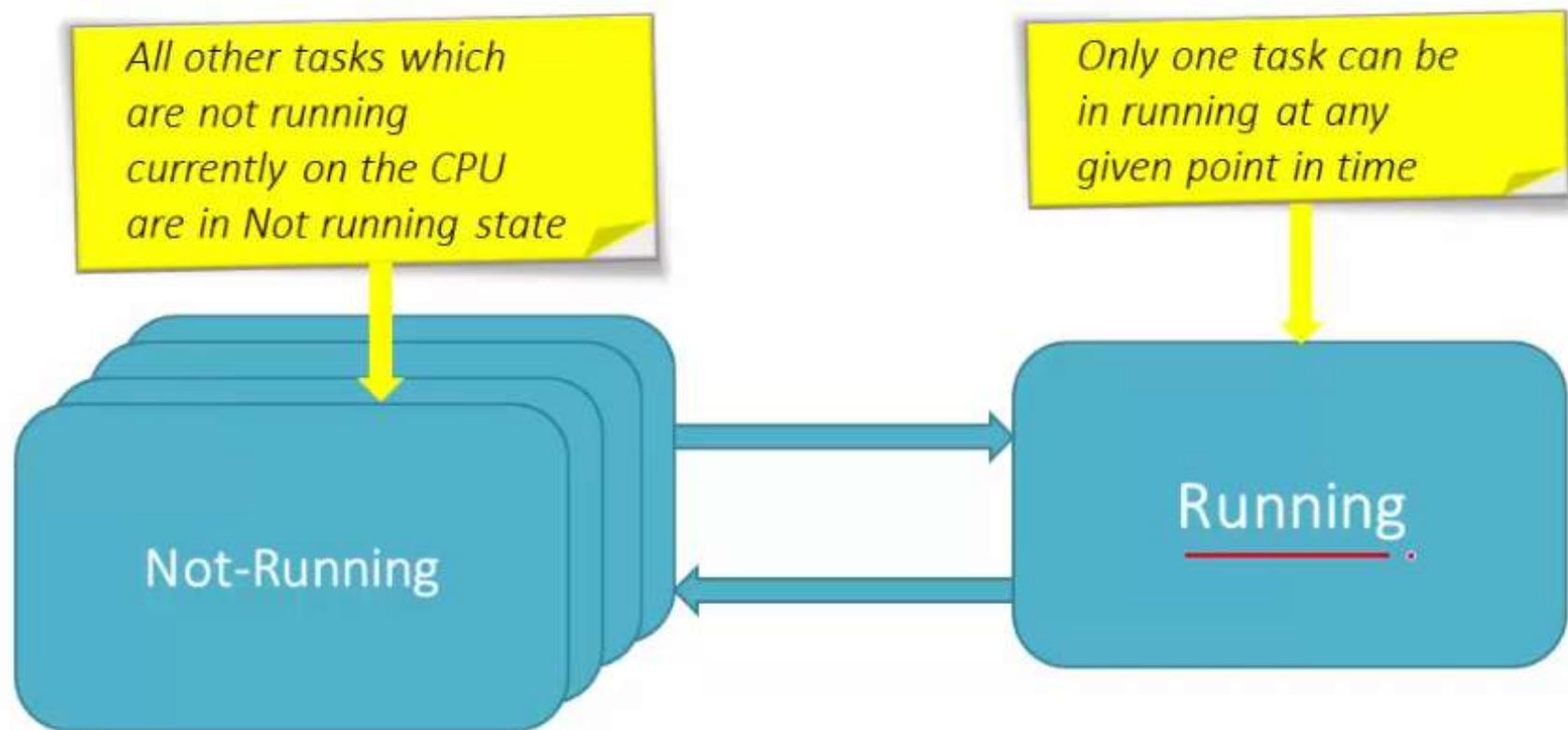
In ARM cortex M based Processors usage exception will be raised by the processor if you return to the task context by preempting ISR .

.

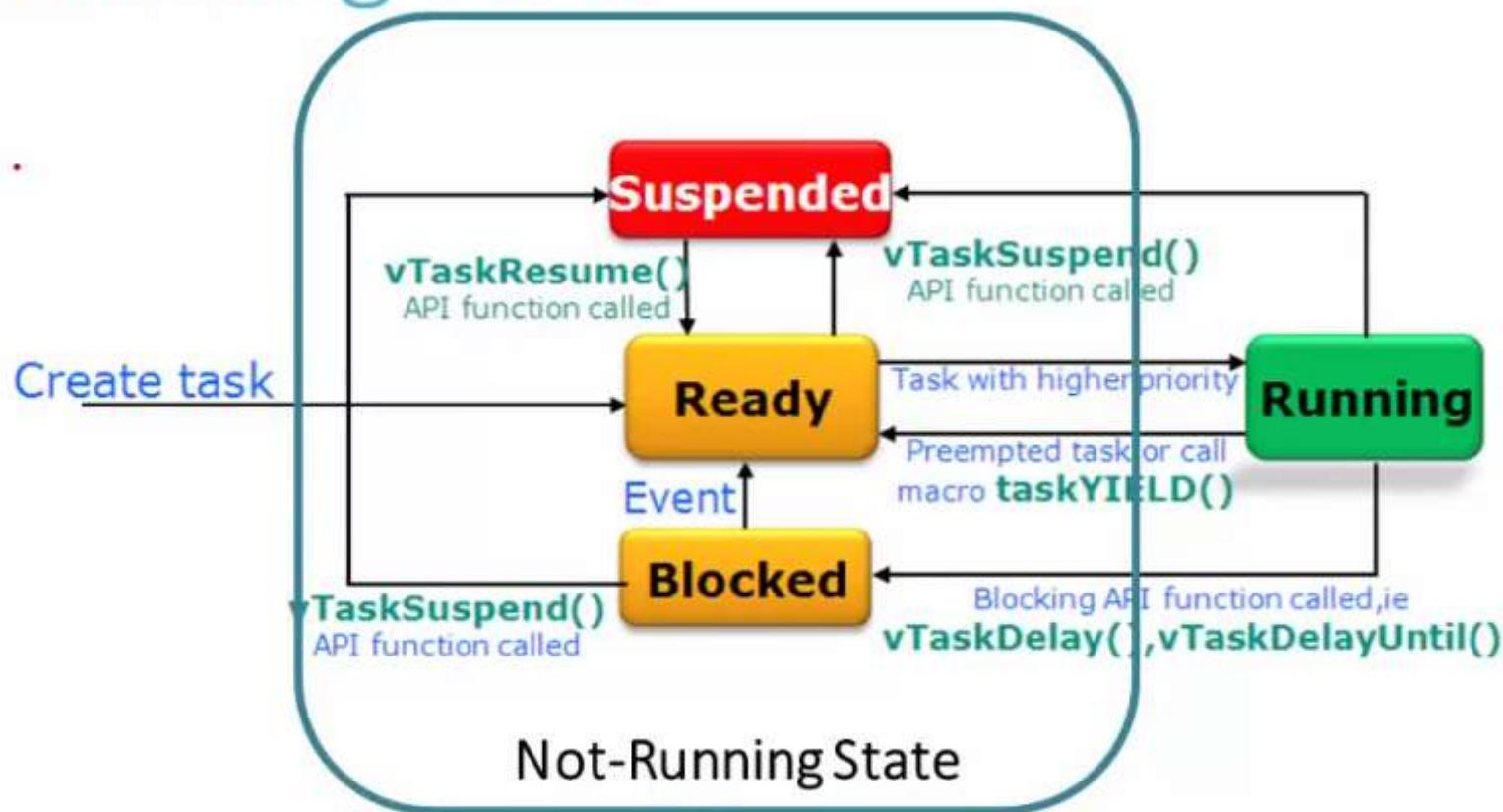
```
7
8  /* Scheduler utilities. */
9
10 #define portYIELD()
11 {
12     /* Set a PendSV to request a context switch. */
13     portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;
14
15     /* Barriers are normally not required but do ensure the code is completely
16      within the specified behaviour for the architecture. */
17     __asm volatile( "dsb" ::: "memory" );
18     __asm volatile( "isb" );
19 }
```

1. Interrupt safe APIs
2. Interrupt Unsafe APIs
3. Why there are separate interrupt safe APIs are provided in freeRTOS
4. Task yielding (i.e Context switching to next potential task)

Top Level Task States- Simplistic Model



Not-Running State



What is blocking of a Task ?

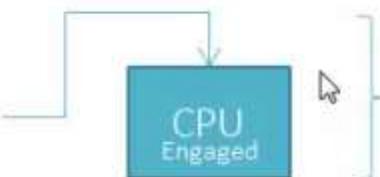


Task is Blocked

A Task which is Temporarily or permanently chosen not to run on CPU

Generate delay of ~ 10ms

```
for ( int i=0 ; i < 5000 ; i++ );
```



This code runs on CPU continuously for 10ms, Starving other lower priority tasks. Never use such delay implementations

vTaskDelay(10)



This is blocking delay API which blocks the task for 10ms. That means for the next 10ms other lower priority tasks can run. After 10ms the task will wake up

Advantages of blocking:

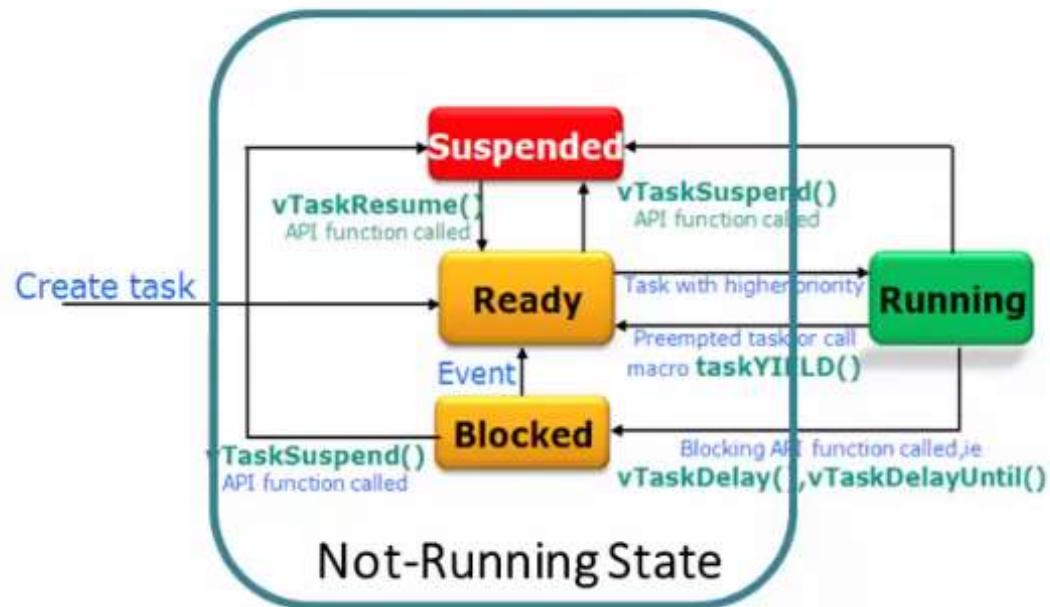
1. To implement the blocking Delay – For example a task may enter the Blocked state to wait for 10 milliseconds to pass.
2. For Synchronization –For example, a task may enter the Blocked state to wait for data to arrive on a queue. When the another task or interrupt fills up the queue , the blocked task will be unblocked.

FreeRTOS queues, binary semaphores, counting semaphores, recursive semaphores and mutexes can all be used to implement synchronization and thus they support blocking of task.

Blocking Delay APIs

vTaskDelay()

vTaskDelayUntil()



Queues

Semaphores

Mutex

All these kernel objects support APIs which can block a task during operation, which we will explore later in their corresponding sections

Suspended State of a Task

Suspended State is also one of the sub-state of not running state .

Most of the applications don't use this state. This is very rarely used task state.

When the task is in suspended state, it is not available to the scheduler to schedule it.

There is only one way to put the task in to suspended state , that is by calling the function `vTaskSuspend()`

The only way to come out from the suspended state is by calling the API `vTaskResume()`

Task States : Conclusion

At any given point, only one task can be in the running state, and other tasks will be in not running state

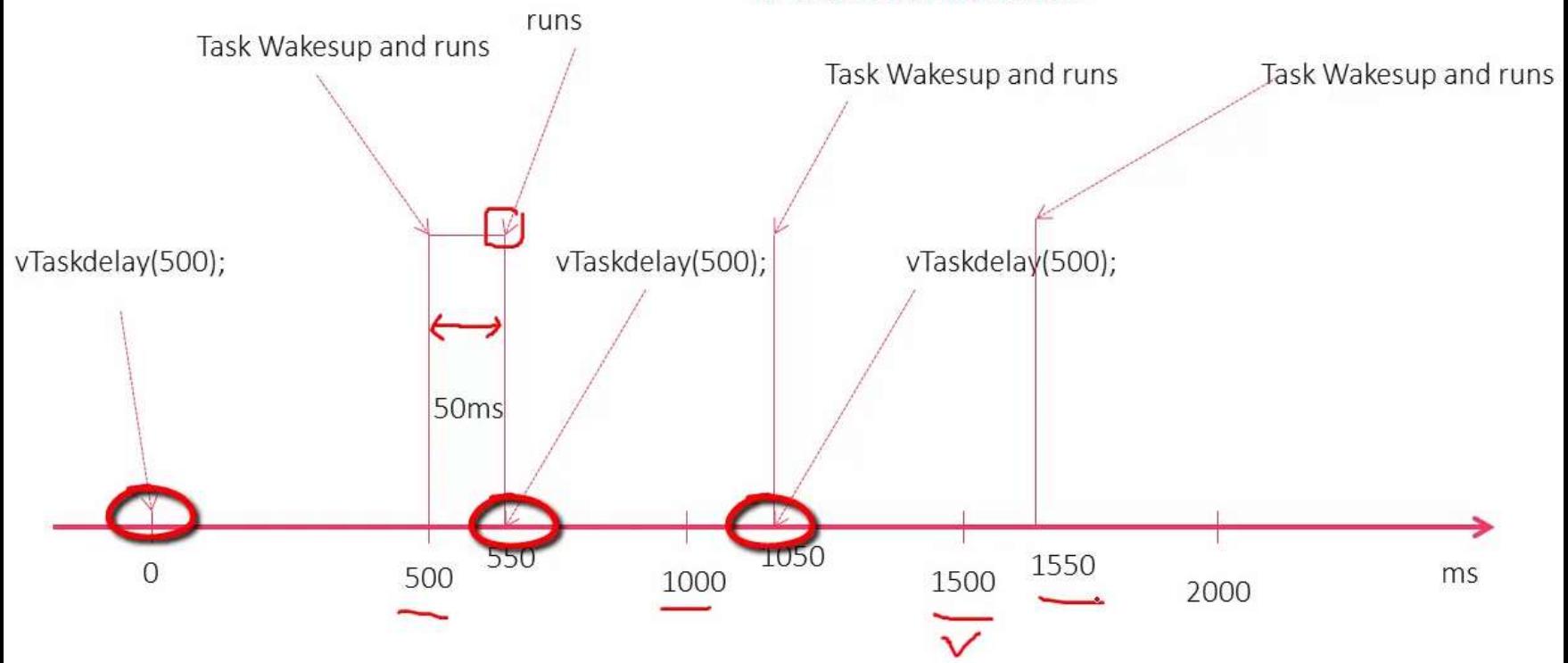
Ready state means, tasks are read to run on the CPU when they get a chance

Blocked state tasks mean, they are temporarily not allowed to run on the CPU, until the unblocking event occurs.

Suspended state means, the tasks are suspended to run on the CPU. 

The only way you can make a task to come out of the suspended state is by calling `vTaskResume()`. No other freeRTOS APIs can move a task from suspended state to ready state.

vTaskDelay() always calculates the time relative to when it was last called.



- Requirement was to wake up the task up at fixed timings, but we missed that, our task wakes up at + 50 ms

vTaskDelayUntil

vTaskDelay() specifies a time at which the task wishes to unblock ***relative to the time at which vTaskDelay() is called***

vTaskDelayUntil() specifies an ***absolute time at which the task wishes to unblock.*** [Time is calculated relative to the last wake up time of the task]

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

- ဒီ api func လေးကတွေ `PreviousWakeTime` ကို
ပြန်ယူထားပါး လိုအပ်တဲ့ အချိန်ကို ပြန်ပီးတိုးပေးမှာဖြစ်တဲ့အတွက်
task တစ်ခုအတွက် delay အတိအကျကိုရရှိနိုင်မှာဖြစ်ပါတယ်။

Convert time to ticks

- Kernel ၏ time ကိုနားမလည်တဲ့အတွက် ထည့်လိုက်တဲ့ time ကို tick ပြန်တွက်ပါတယ်။
- configTICK_RATE_HZ = 1000 Hz သည် kernel အတွက်ဖြစ်ပါတယ်။
1000 Tick interrupt မှာ 1 sec နဲ့ညီမျှပါတယ်။
- TICK_RATE_MS = $1/1000 = 1\text{ms}$
- အူးအတွက် 2 ticks အင်တာရပ် နှစ်ခုကြားမှာ 1 ms ကြာမှာပါ။
- အုပော - 250 Hz ကို သတ်မှတ်ထားမယ်ဆွင် $1/250 = 4 \text{ ms}$ ကြာမှာပါ။

Task 1 prints out its string ,
then it too enters the
Blocked state by calling
vTaskDelay()

2

When the delay expires the scheduler moves the
tasks back into the ready state, where both
execute again before once again calling
vTaskDelay() causing them to re-enter the blocked
state. Task-2 executes first as it has the higher
Priority

4

Task 1

Task 2

Idle

t1

t2

tn

Time

1

Task 2 has the highest priority so runs first. It
prints out its string then calls vTaskDelay() and in
so doing enters the blocked state, permitting the
lower priority Task 1 to execute.

3

At this point both applications tasks are
in the blocked state-so the idle task
runs

အရောက်ပုံရဲရင်းပြချက်

- Task 2 ၏ priority ပိုမြင့်တဲ့အတွက် အရင်ဆုံး run ပါတယ်။
- Common function တစ်ခုထဲကို Task 1 ကပါ ထပ် run ပါမယ်။
- Common func မှာ delay သုံးထားတဲ့အတွက် task နှစ်ခုလုံး run ပြီးသွားတာနဲ့ idle state ကြီး ဝင်သွားမှာဖြစ်တယ်။

