Real time operating system

Free RTOS & CMSIS RTOS V2

STM 32

Real Time Operating System(RTOS)

Myth

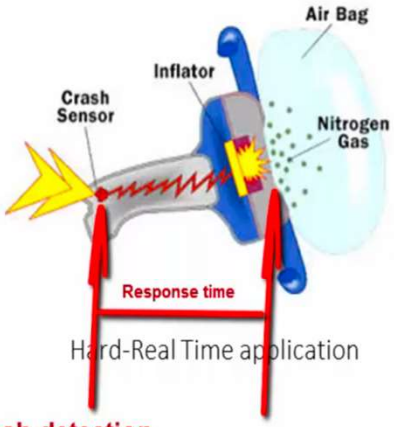
- Real-time computing is equivalent to fast computing.
- Higher performance.

Truth

- The real-time computing is equivalent to predictable computing.
- Computing timeliness is more important than performance.
- Guarantee , not raw speed.

RTOS(Cont'd)

- Having more Processors, more RAM, faster bus interfaces doesn't make a system real time.!!
- A real time system deals with guarantees, not with promises.
- A real time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation, but also upon the at which the result is produced.If the timing constraints are not met,system failure is said to have occurred.



Hard-Real Time application

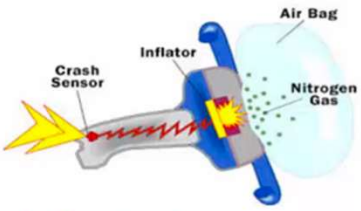
Crash detection

Air bag deployment

May be soft-real time application ?
Delay is tolerable

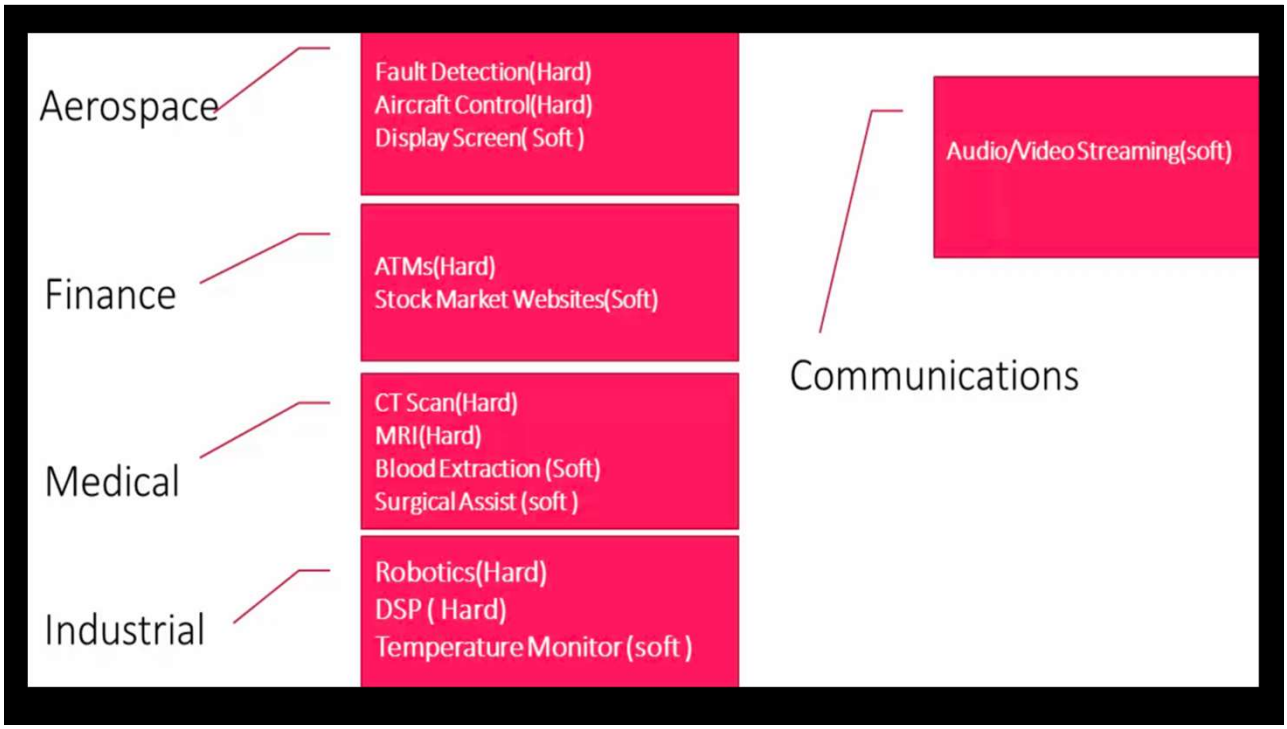
Missile guidance and control systems

Ani-Lock Breaking System



Airbag Deployment

Stock Market Website



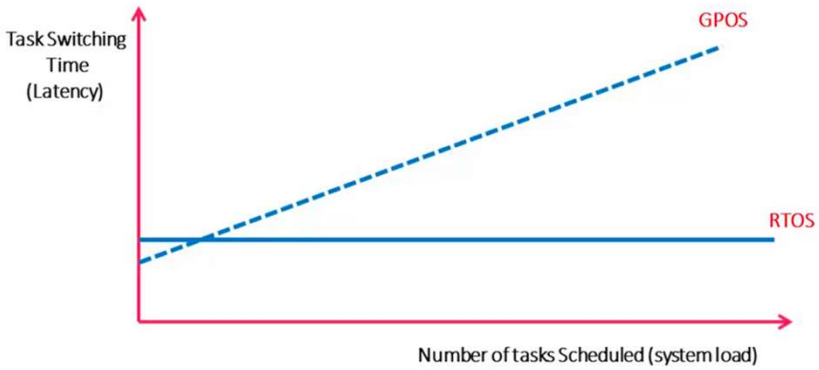
High throughput

- Fairness policy
- Dispatch threads and process

Poor throughput

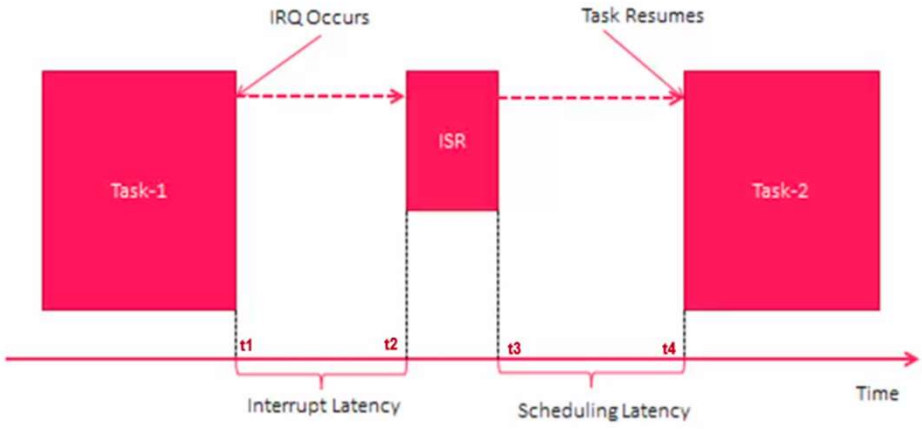
- Time Deterministic
- Guarantee (not promise)
- Predictability
- Latency (time that elapsed between a stimulus and the response to it.)
- Task switching latency (the time gap between "A triggering of an event and a time at which the task which take care of that event is allowed to run on the CPU")

RTOS vs GPOS: Task Switching Latency

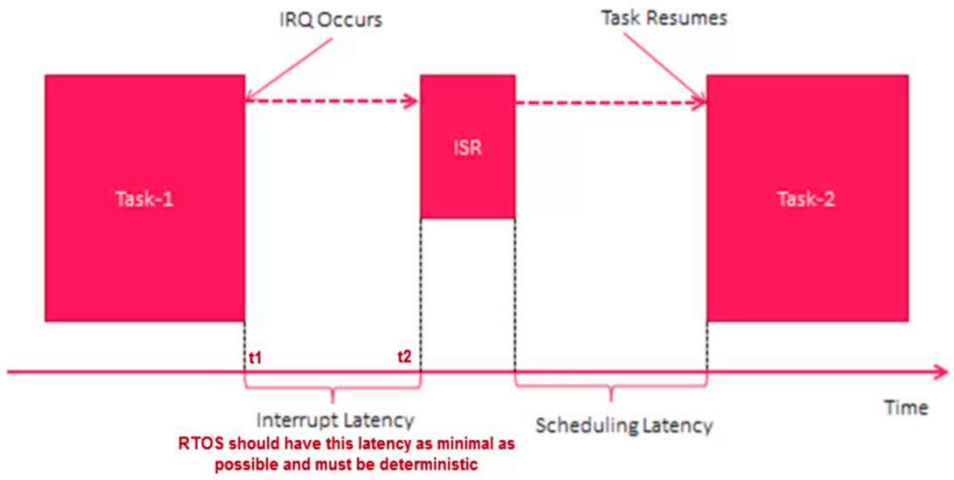


Task switching latency remains almost constant in RTOS.
May vary significantly on GPOS.

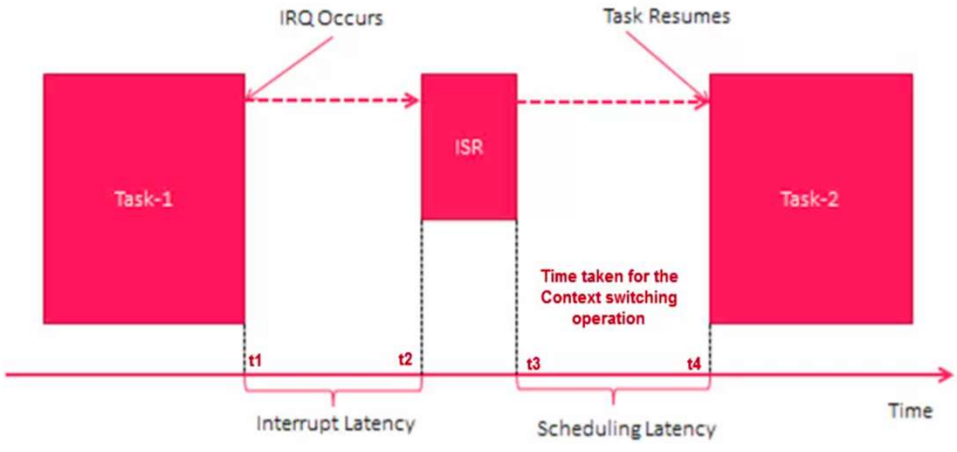
RTOS vs GPOS: Interrupt Latency



RTOS vs GPOS: Interrupt Latency



RTOS vs GPOS: Interrupt Latency



Preempt

- Both the interrupt latency and scheduling latency of the RTOS is small as possible and time bounded.
- But in the case of GPOS, due to increase in system load these parameters may vary significantly.

Scheduling Policies (Scheduler types)

1. simple Pre-emptive Scheduling (Round robin)
2. Priority based Pre-Emptive Scheduling
4. Co-operative Scheduling

The **scheduling policy** is the algorithm used by the scheduler to decide which task to execute at any point in time.

FreeRTOS or most of the Real Time OS most likely would be **using Priority based Pre-emptive Scheduling** by default

Stack and Heap in embedded Systems

FILO

SP

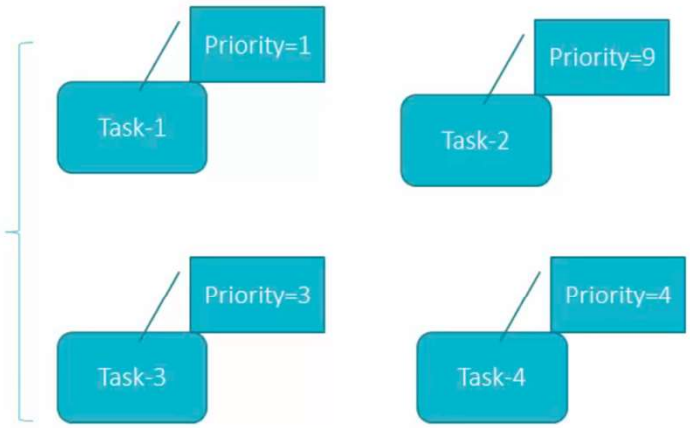
Stack access is automatically tracked by SP register of your MCU.

Fist in Last out Order Access

HP

out of order access of memory

Scheduler

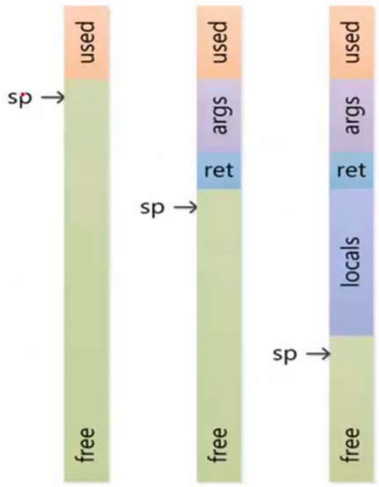


I have my own **scheduling policy** , I act according that .

You can configure **my scheduling policy**

Task List

Stack



sp : Stack Pointer
used : Unavailable stack (already used)
args : function arguments
ret : return address
locals : local variable
free : available stack

Heap

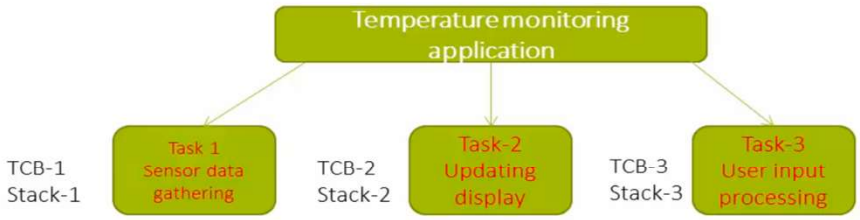
A heap is a general term used for any memory that is allocated dynamically and randomly.

I understand stack is managed by SP and dedicated instructions like PUSH n POP , how Heap is managed ?

How the heap is managed is really up to the runtime environment. C uses **malloc** and C++ uses **new**.

But for embedded systems malloc and free APIs are not suitable because they eat up large code space , lack of deterministic nature and fragmented over time as blocks of memory are allocated

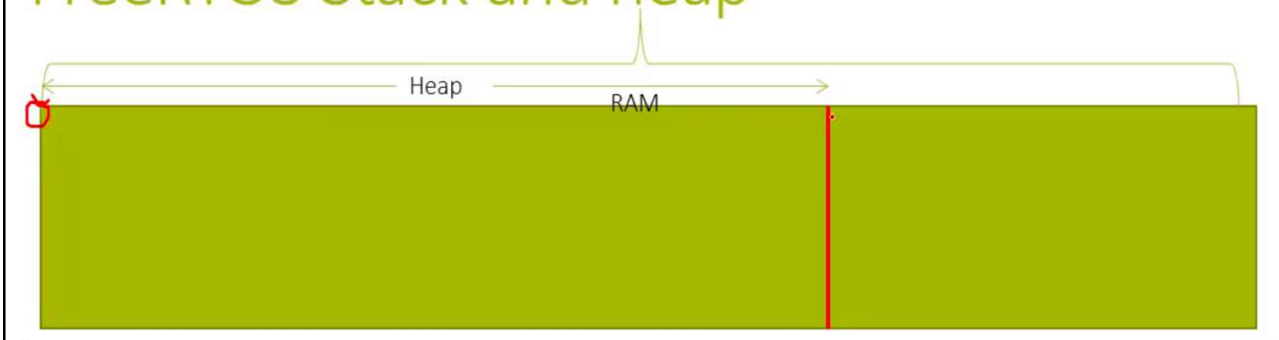
FreeRTOS Stack and heap management



Where do you think task's TCB and its associated stack will be created?

There are 2 options
1) if you use dynamic creation method then they will be created in the heap memory of the RAM
2) if you create them statically then they will be created in other part of the RAM except heap and stack space

FreeRTOS Stack and heap

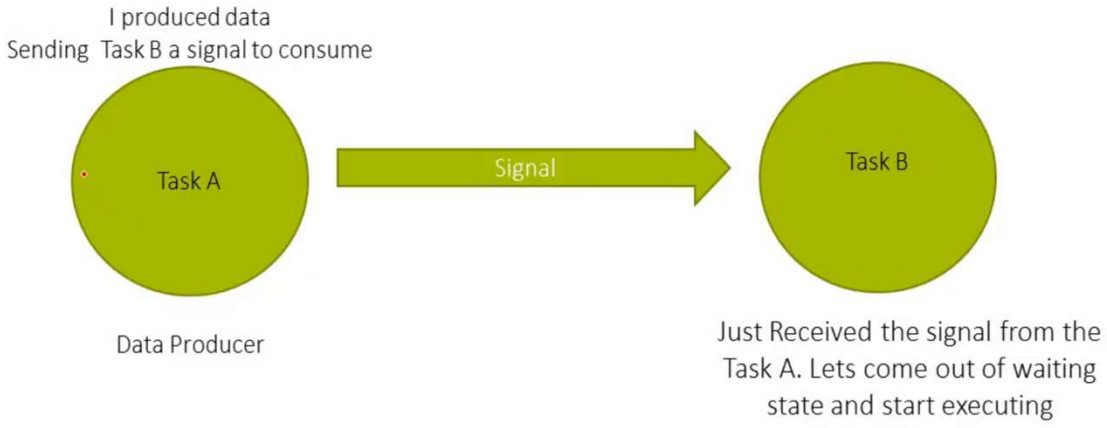


Who decides the starting address and size of the heap?

By default the FreeRTOS heap is declared by FreeRTOS kernel

Setting configAPPLICATION_ALLOCATED_HEAP to 1 allows the heap to instead be declared by the application

Synchronization between Tasks



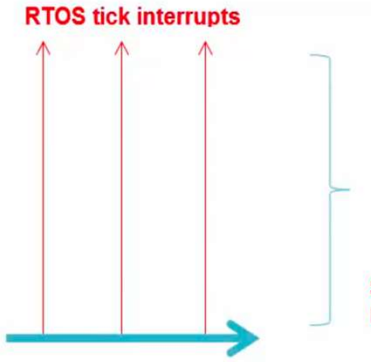
How to achieve this signaling ?

Events (or Event Flags)
Semaphores (Counting and binary)
Queues and Message Queues
Pipes
Mailboxes
Signals (UNIX like signals)
Mutex

All these software subsystems support signaling hence can be used in Synchronization purposes

The RTOS Tick- Why it is needed ?

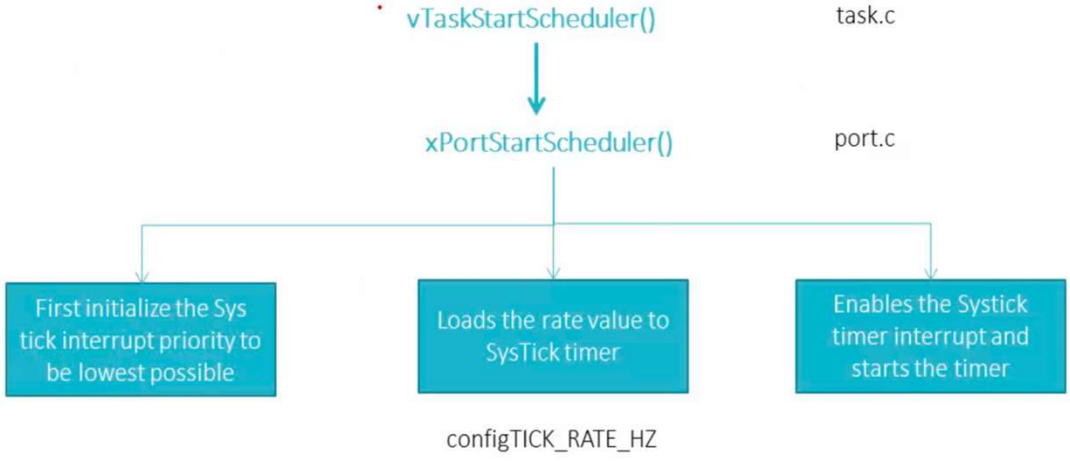
Used for Context Switching to the next potential Task



1. The tick ISR runs
2. All the ready state tasks are scanned
3. Determines which the next potential task to run
4. If found, triggers the context switching handler

So, the Tick ISR doesn't carry out Context switching , it just invokes another handler that later will carry out the context switching operation !

Who configures OS tick Timer ?

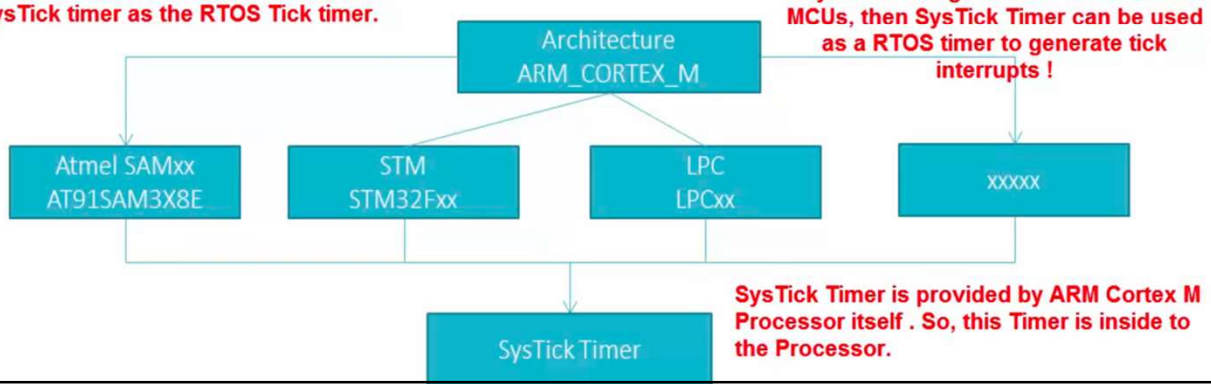


Does kernel use any timer Peripheral to issue tick interrupts ?

YES !! Of course, It should be implemented using timer hardware of the MCU

Remember!! All the official FreeRTOS, ARM Cortex M specific codes will consider sysTick timer as the RTOS Tick timer.

if you are using ARM Cortex M based MCUs, then SysTick Timer can be used as a RTOS timer to generate tick interrupts !



SysTick Timer is provided by ARM Cortex M Processor itself . So, this Timer is inside to the Processor.

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

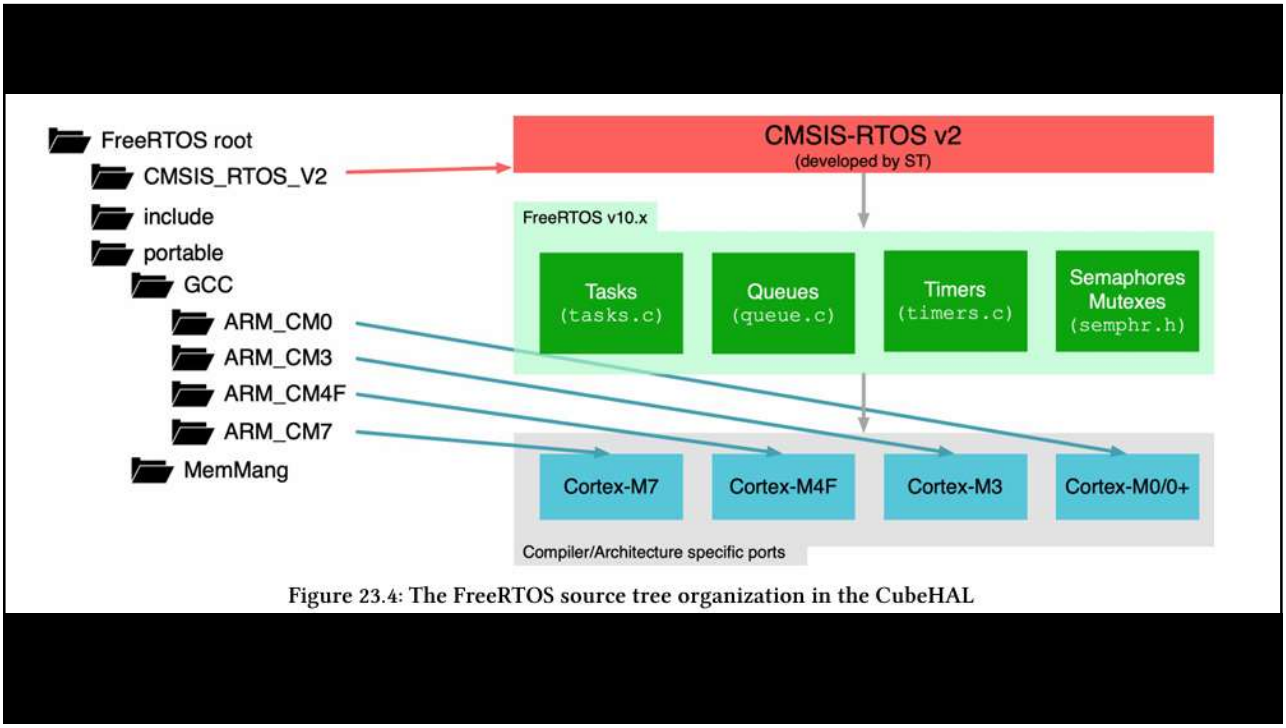
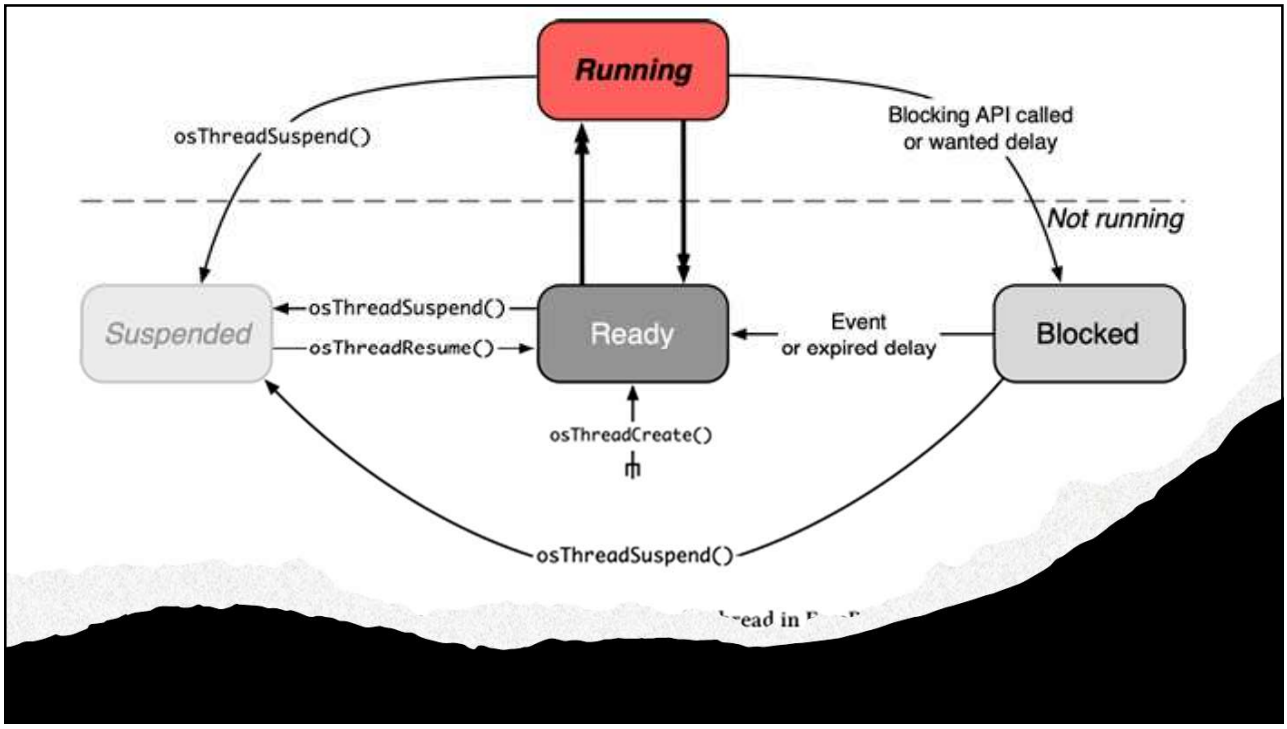


Figure 23.4: The FreeRTOS source tree organization in the CubeHAL

Thread Management

- Thread States
- Thread Priorities and Scheduling Policies
- Voluntary Release of the Control
- The *idle* thread



Priority level	Value	Description
osPriorityNone	0	No priority (not initialized).
osPriorityIdle	1	Priority: *idle* - Reserved for Idle thread.
osPriorityLow	8	Priority: *low*
osPriorityLow[1..7]	8+[1..7]	Priority: low + 1..7
osPriorityBelowNormal	16	Priority: *below normal*
osPriorityBelowNormal[1..7]	16+1	Priority: below normal + 1..7
osPriorityNormal	24	Priority: *normal*
osPriorityNormal[1..7]	24+1	Priority: normal + 1..7
osPriorityAboveNormal	32	Priority: *above normal*
osPriorityAboveNormal[1..7]	32+1	Priority: above normal + 1
osPriorityHigh	40	Priority: *high*
osPriorityHigh[1..7]	40+1	Priority: high + 1
osPriorityRealtime	48	Priority: *realtime*
osPriorityRealtime[1..7]	48+1	Priority: realtime + 1
osPriorityISR	56	Priority: *ISR* - Reserved for ISR deferred thread
osPriorityError	-1	System cannot determine priority or illegal priority
osPriorityReserved	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'*

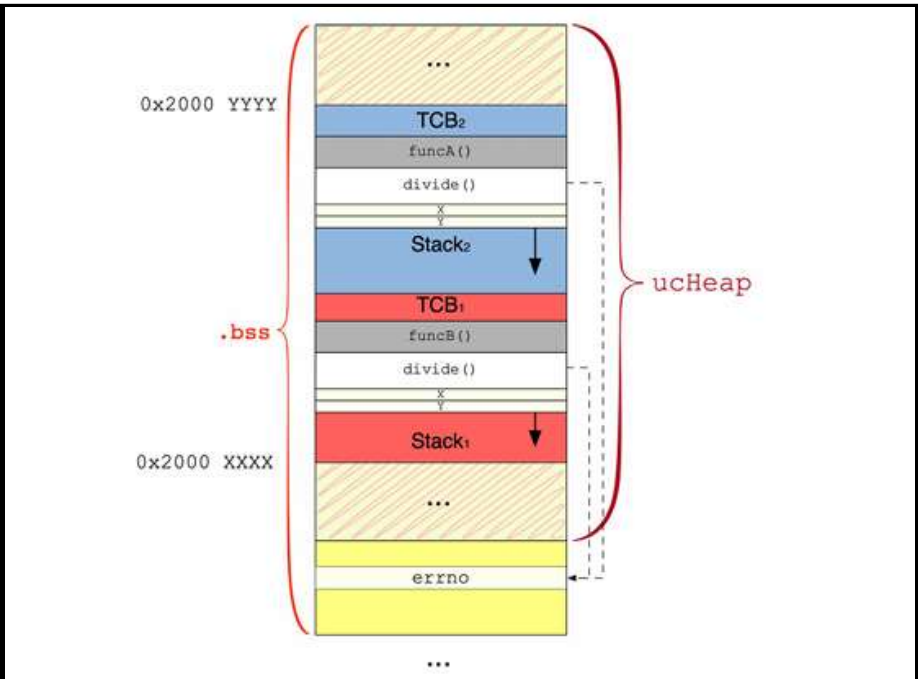


Figure 23.10: The memory layout of the `divide()` function in a multi-threaded environment

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()`.

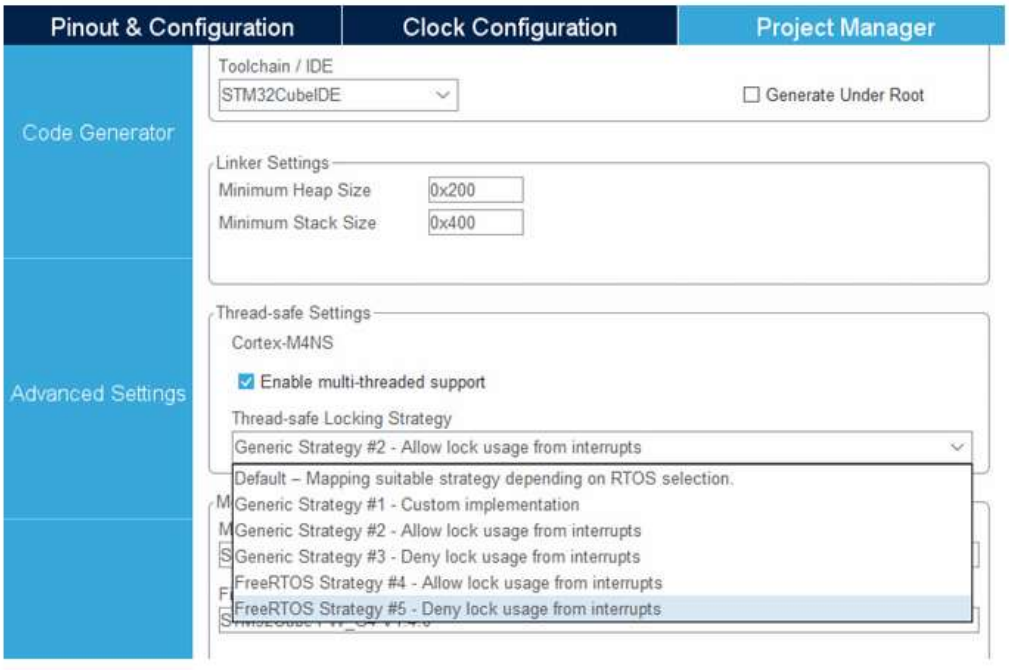


Figure 23.14: How to select Thread-Safe Locking Strategy in CubeMX

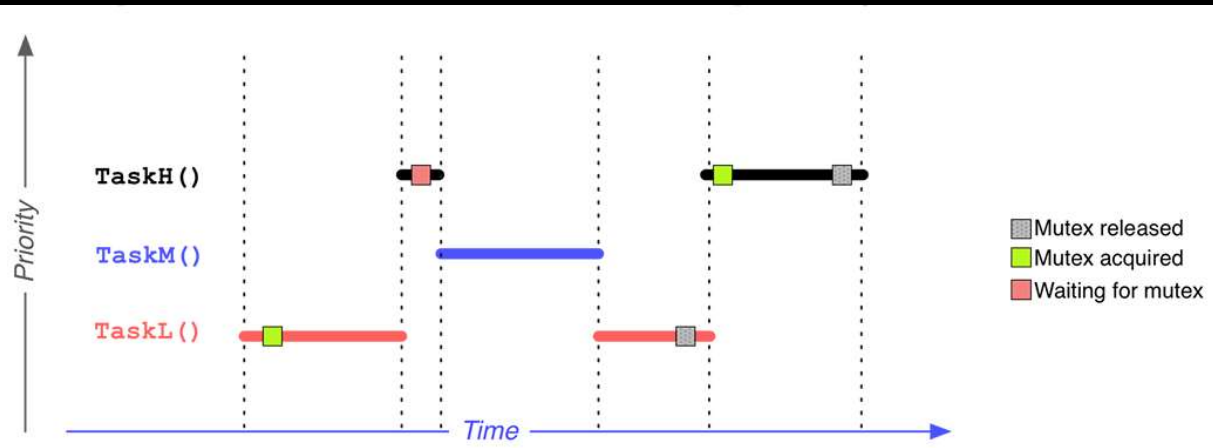


Figure 23.19: The diagram schematizes the *priority inversion* problem

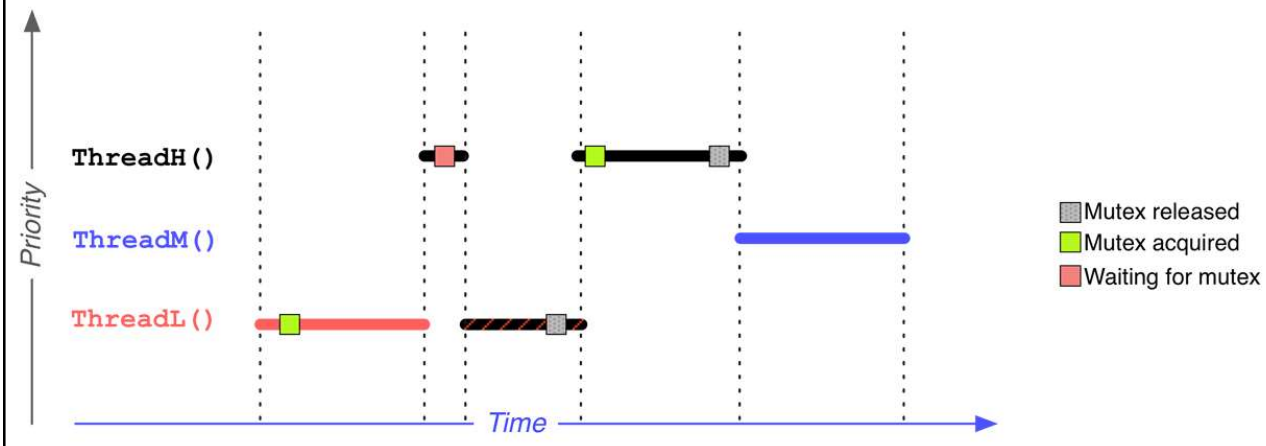
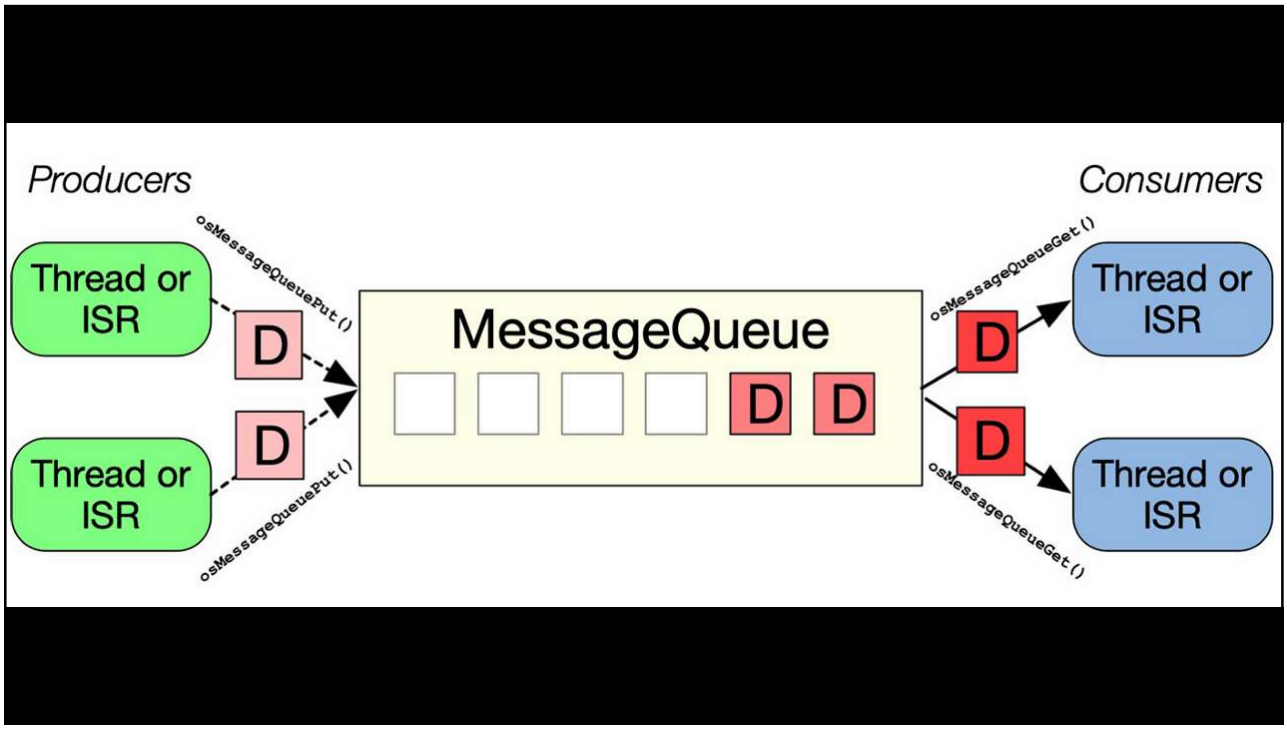
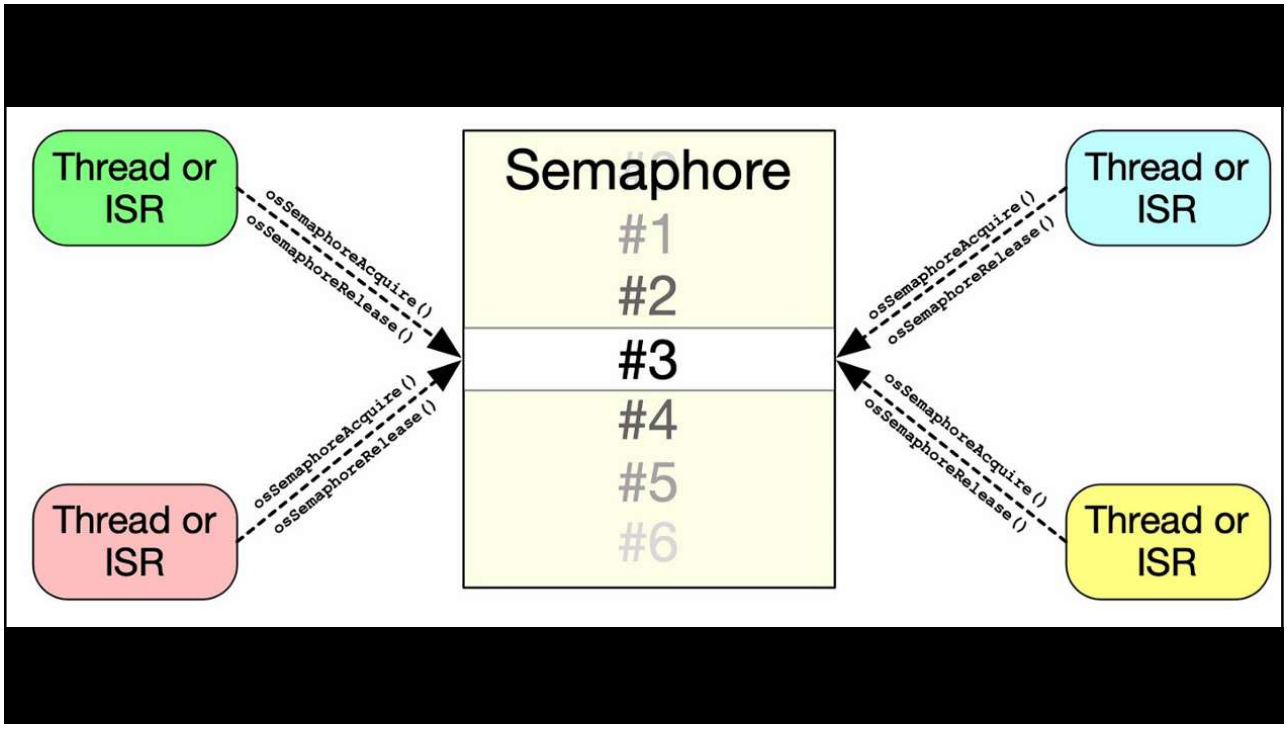
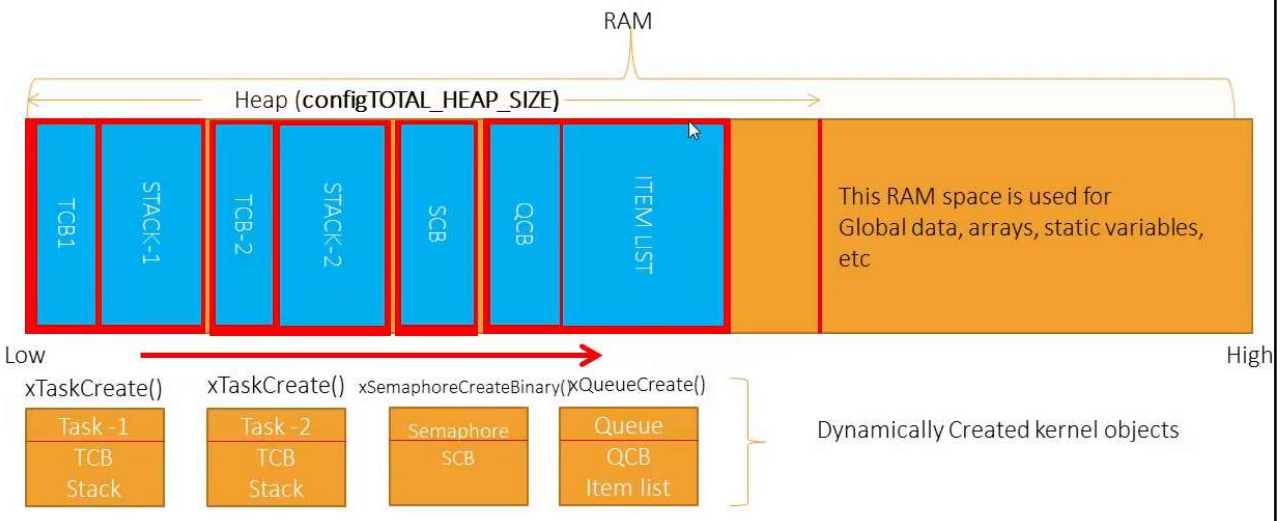


Figure 23.20: How the *priority inversion* problem is addressed by temporary increasing the priority of `ThreadL`

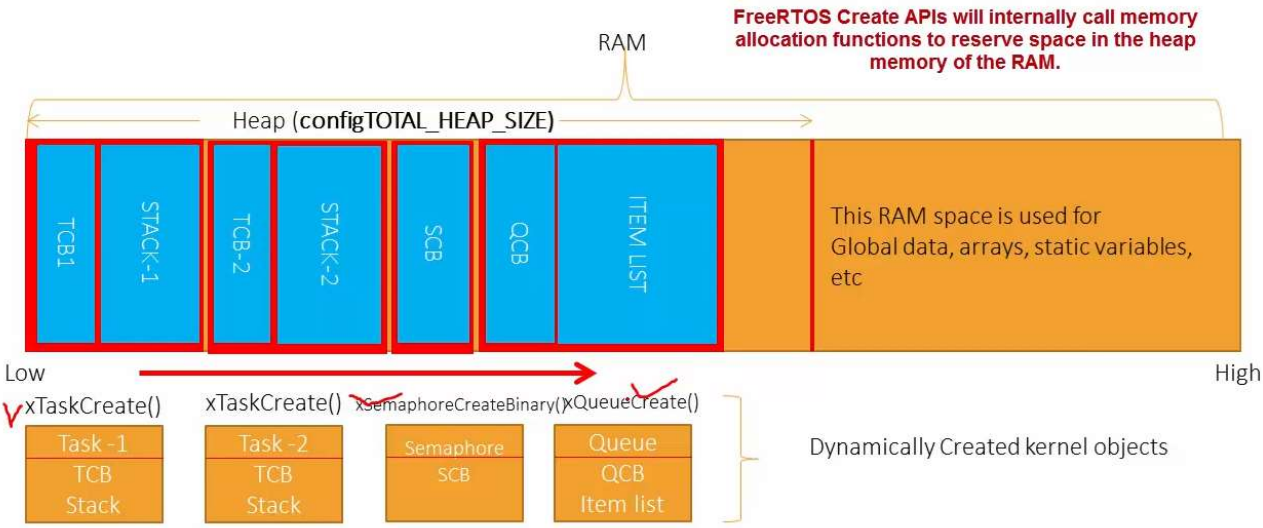

Producers
Consumers
Thread or ISR
Thread or ISR
D
D
MessageQueue
D
D
D
D
Thread or ISR
Thread or ISR
osMessageQueuePut()
osMessageQueuePut()
osMessageQueueGet()
osMessageQueueGet()


Thread or ISR
Thread or ISR
Semaphore
#1
#2
#3
#4
#5
#6
Thread or ISR
Thread or ISR
osSemaphoreAcquire()
osSemaphoreRelease()
osSemaphoreAcquire()
osSemaphoreRelease()
osSemaphoreAcquire()
osSemaphoreRelease()
osSemaphoreAcquire()
osSemaphoreRelease()

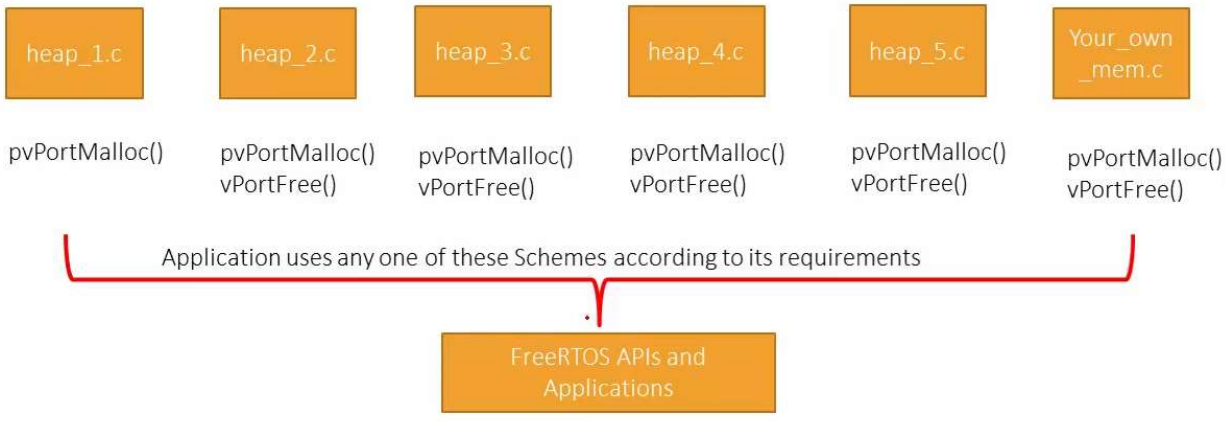
FreeRTOS Stack and heap



FreeRTOS Stack and heap



FreeRTOS Heap management Scheme



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.

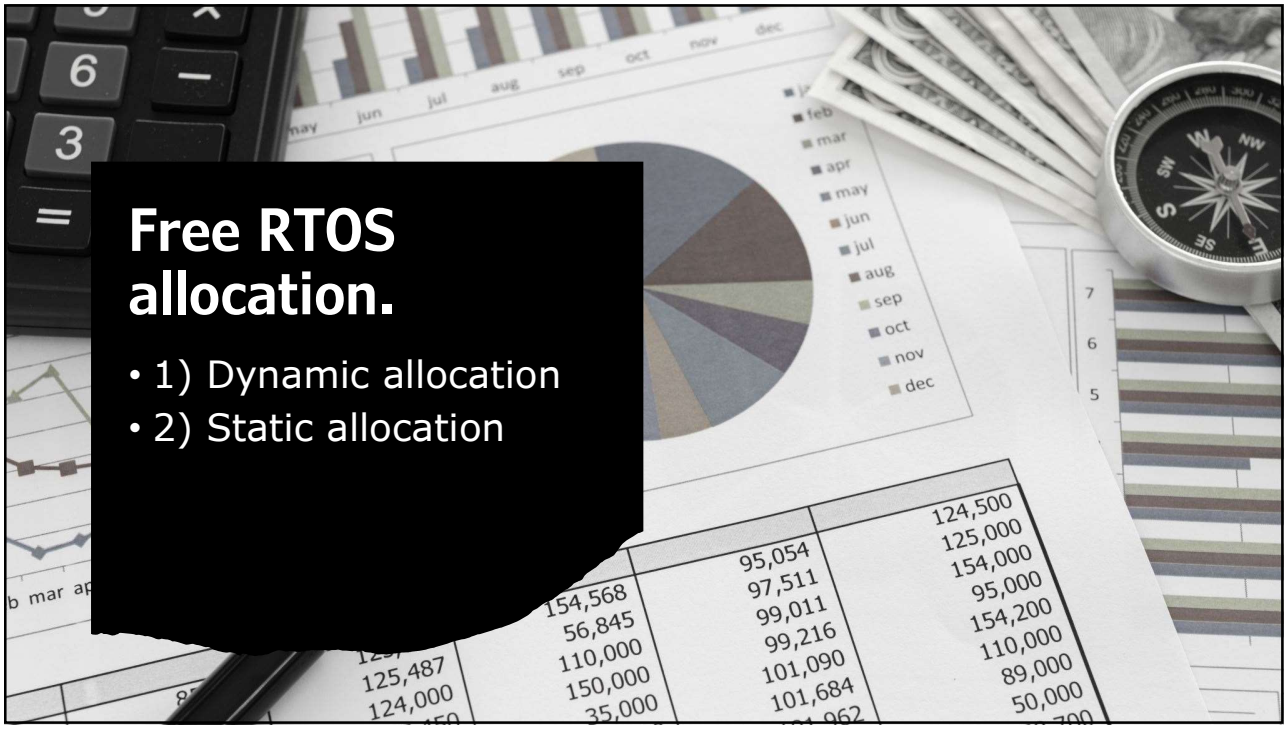
Heap 5.c

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.



Free RTOS 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 libray 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.

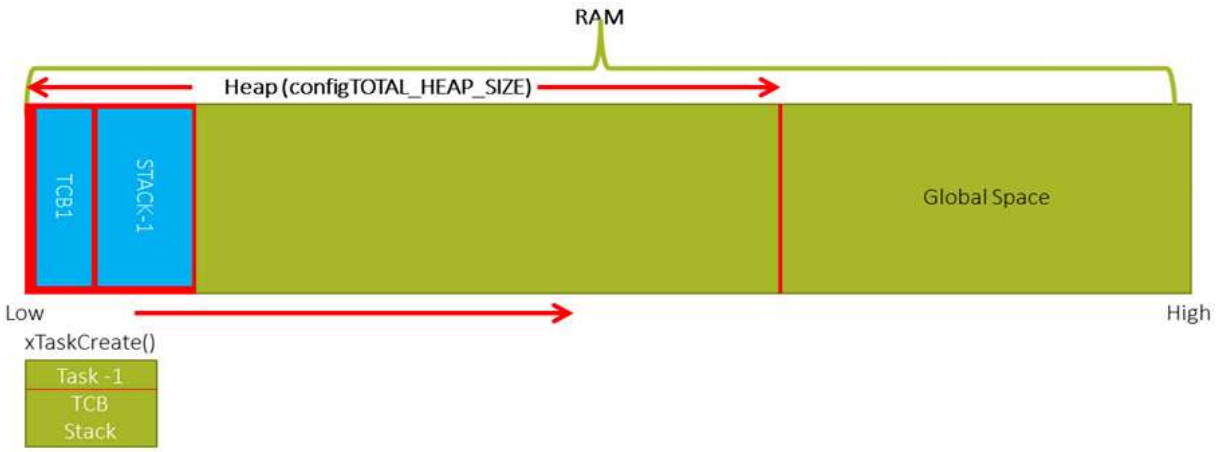
Memory Management scheme

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)

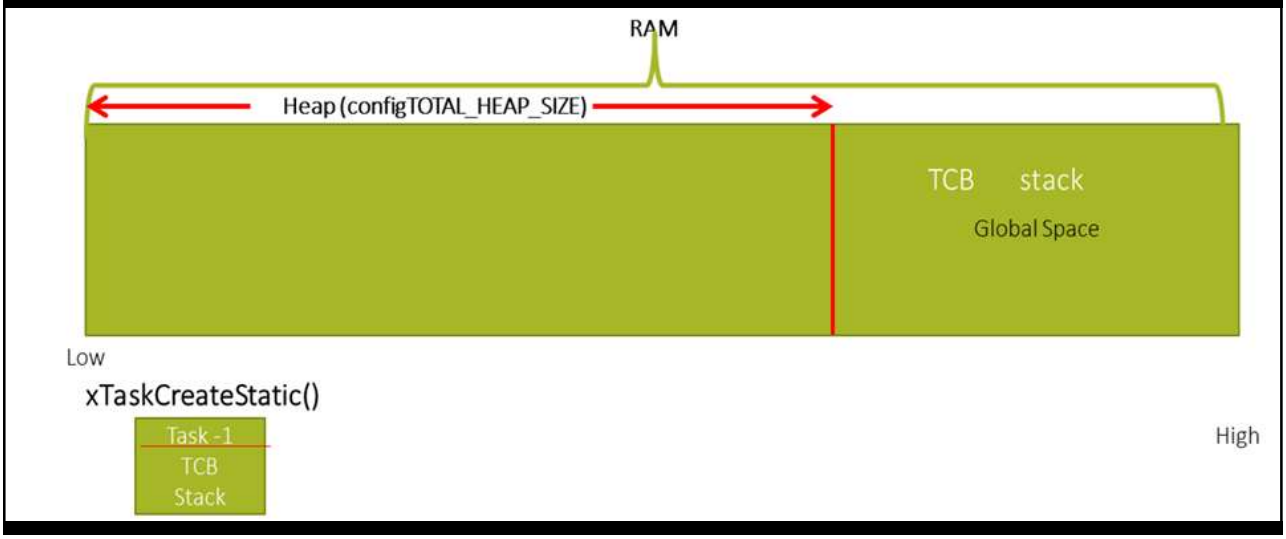
How do you take some part of the RAM? Use Array!!!!

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)



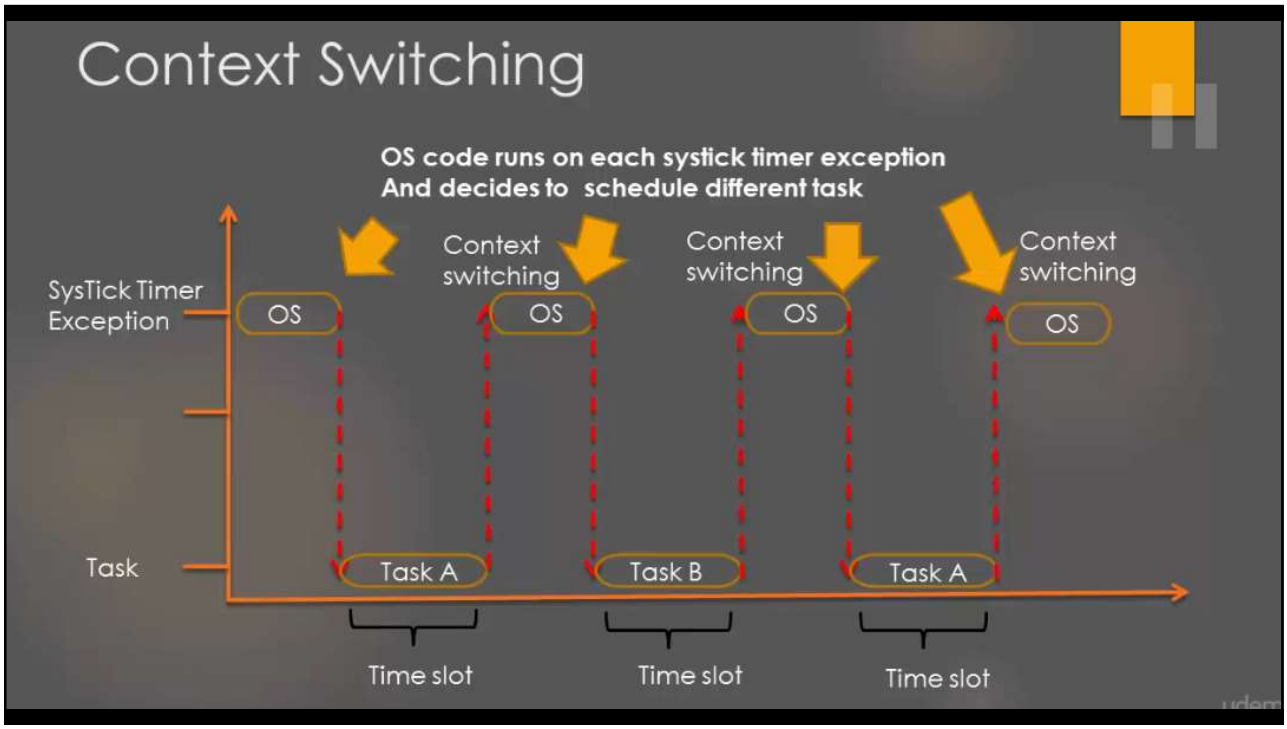
ARM Cortex M Architecture Specific code mainly consists of

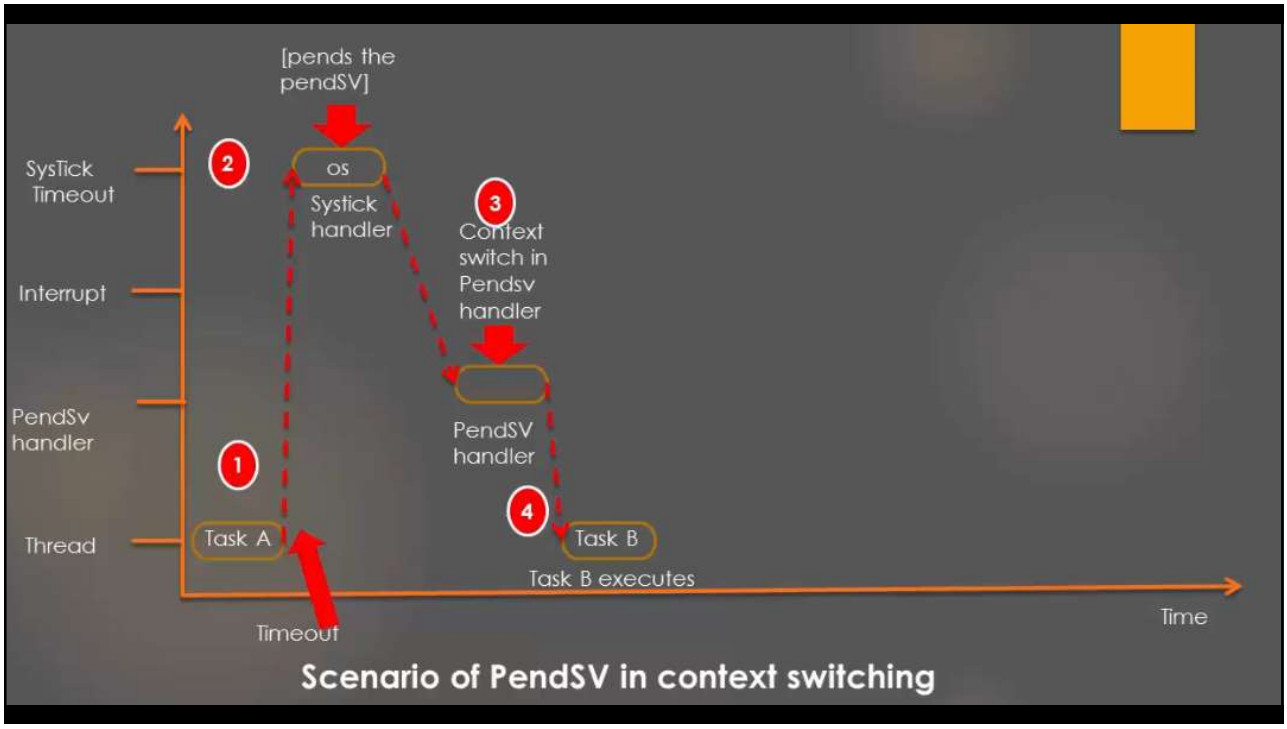
1. SVC Handler (for Scheduling first task)
2. PendSV Handler (for Context Switching)
3. SysTick Handler (for OS ticking)

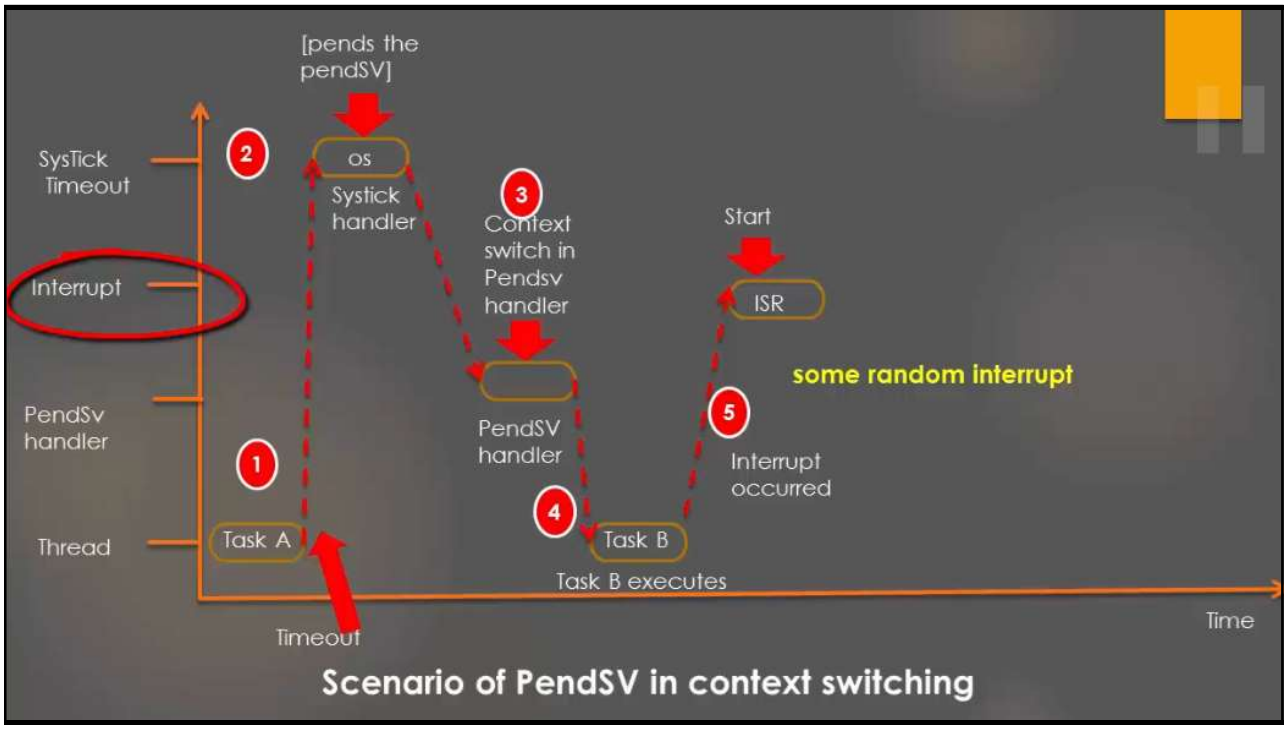
Pend SV Handler

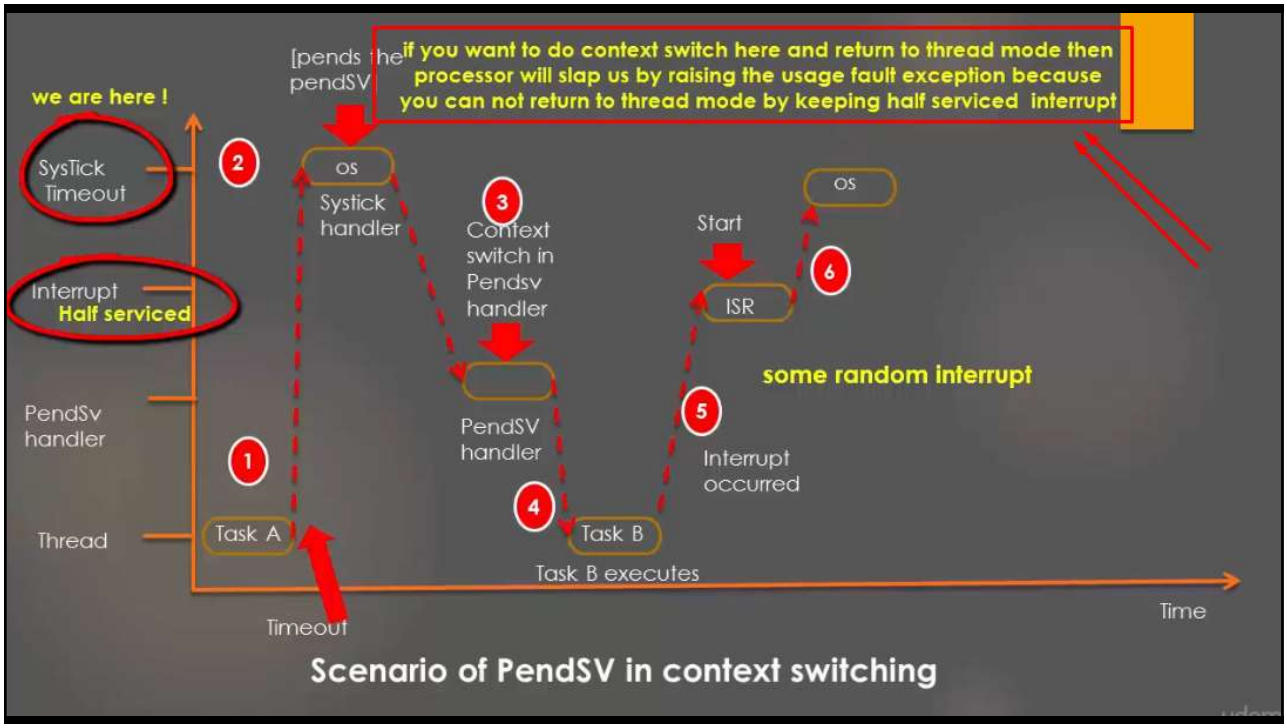
PendSV Exception

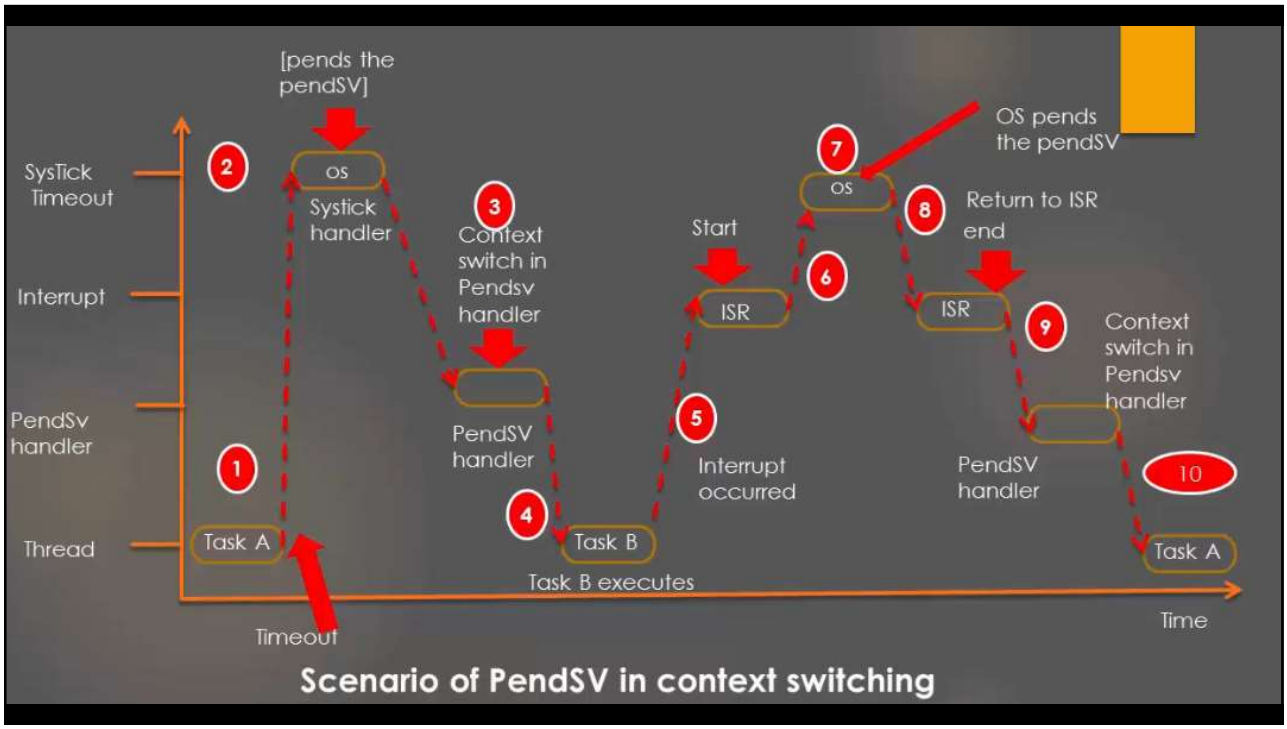
- In OS Designs, we need to switch between different tasks to support multitasking. This is typically called context switching.
- Context switching is usually carried out inside the PendSV exception handler.
- It is exception type 14, mean exception number and has a programmable priority level.
- possible 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
OS code runs on each systick timer exception
And decides to schedule different task
SysTick Timer Exception
Task
OS
Context switching
Task A
Task B
Time slot

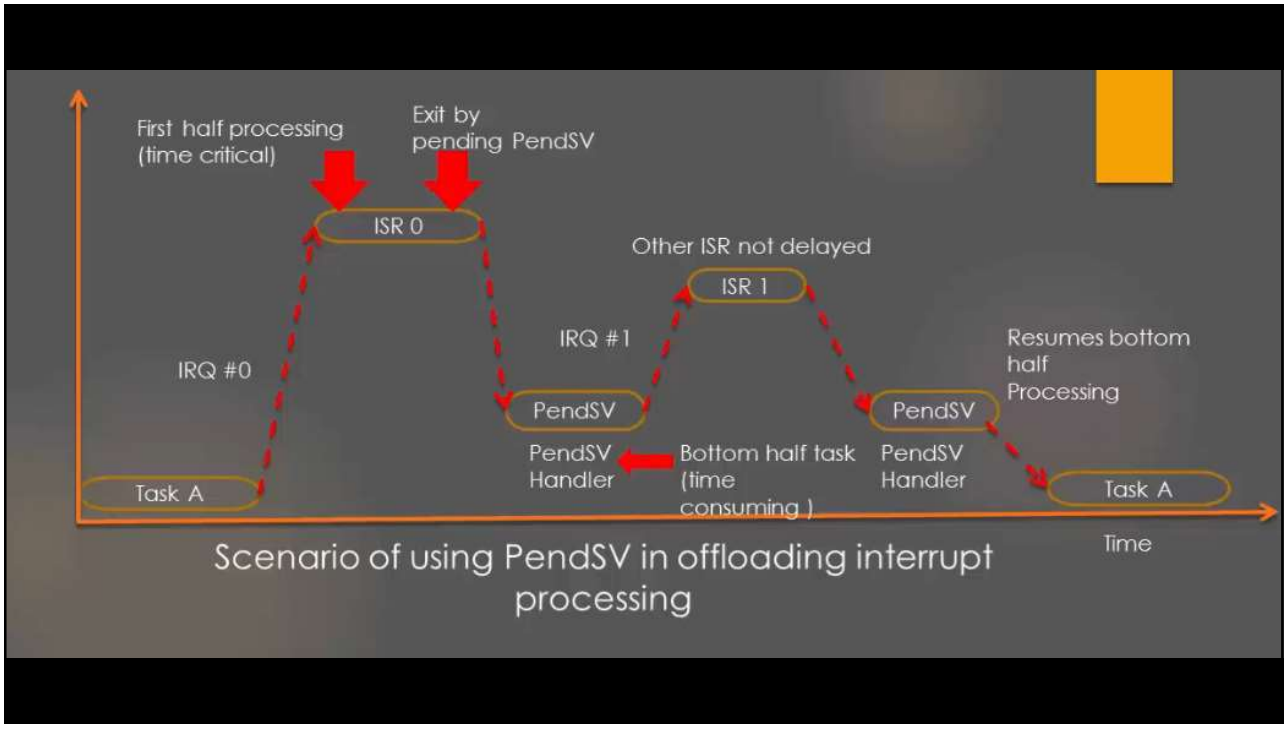

[pends the pendSV]
SysTick Timeout
Interrupt
PendSv handler
Thread
os
Systick handler
Context switch in Pendsv handler
PendSV handler
Task A
Task B
Task B executes
Timeout
Time
Scenario of PendSV in context switching


[pends the pendSV]
SysTick Timeout
2
os
Systick handler
3
Context switch in Pendsv handler
Start
ISR
Interrupt
some random interrupt
PendSv handler
PendSV handler
5
Interrupt occurred
1
4
Thread
Task A
Task B
Task B executes
Timeout
Time
Scenario of PendSV in context switching


we are here !
[pends the pendSV
if you want to do context switch here and return to thread mode then processor will slap us by raising the usage fault exception because you can not return to thread mode by keeping half serviced interrupt
SysTick Timeout
2
os
Systick handler
3
Context switch in Pendsv handler
Start
os
6
ISR
Interrupt
Half serviced
some random interrupt
PendSv handler
PendSV handler
5
Interrupt occurred
1
4
Thread
Task A
Task B
Task B executes
Timeout
Time
Scenario of PendSV in context switching

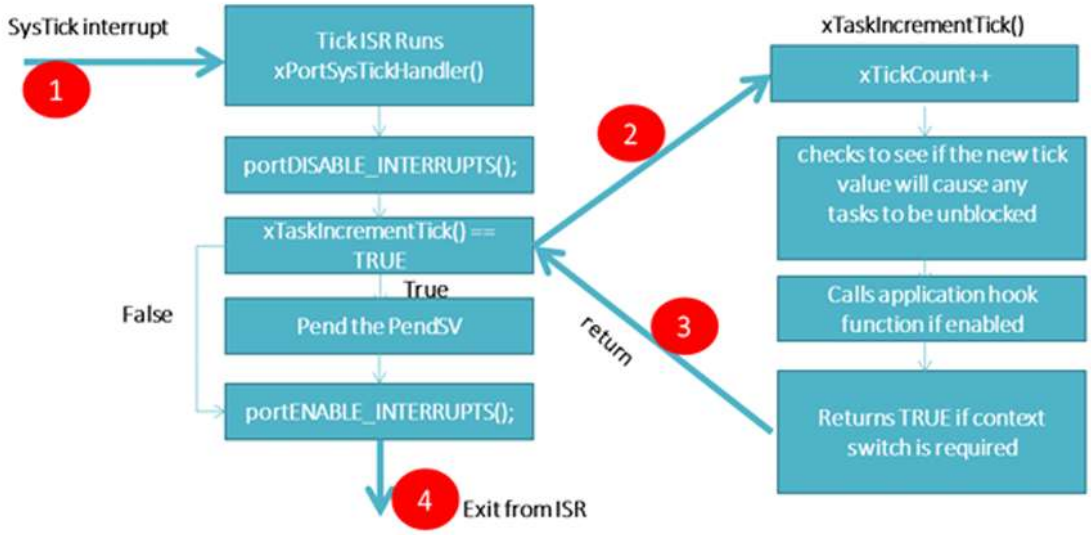


Scenario of PendSV in context switching



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.

.

WARNING

In order to ensure that **sysclk_get_cpu_hz()** return the correct system frequency, the **"sysclck_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.

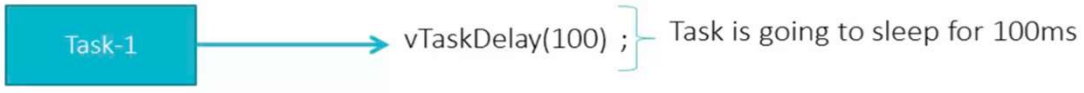
The frequency of the RTOS tick interrupt.

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 ?

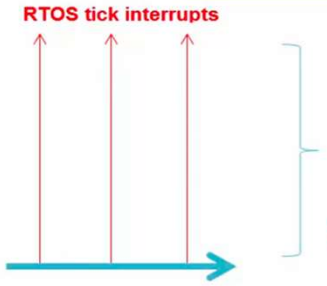
Used for Context Switching to the next potential Task

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 ?

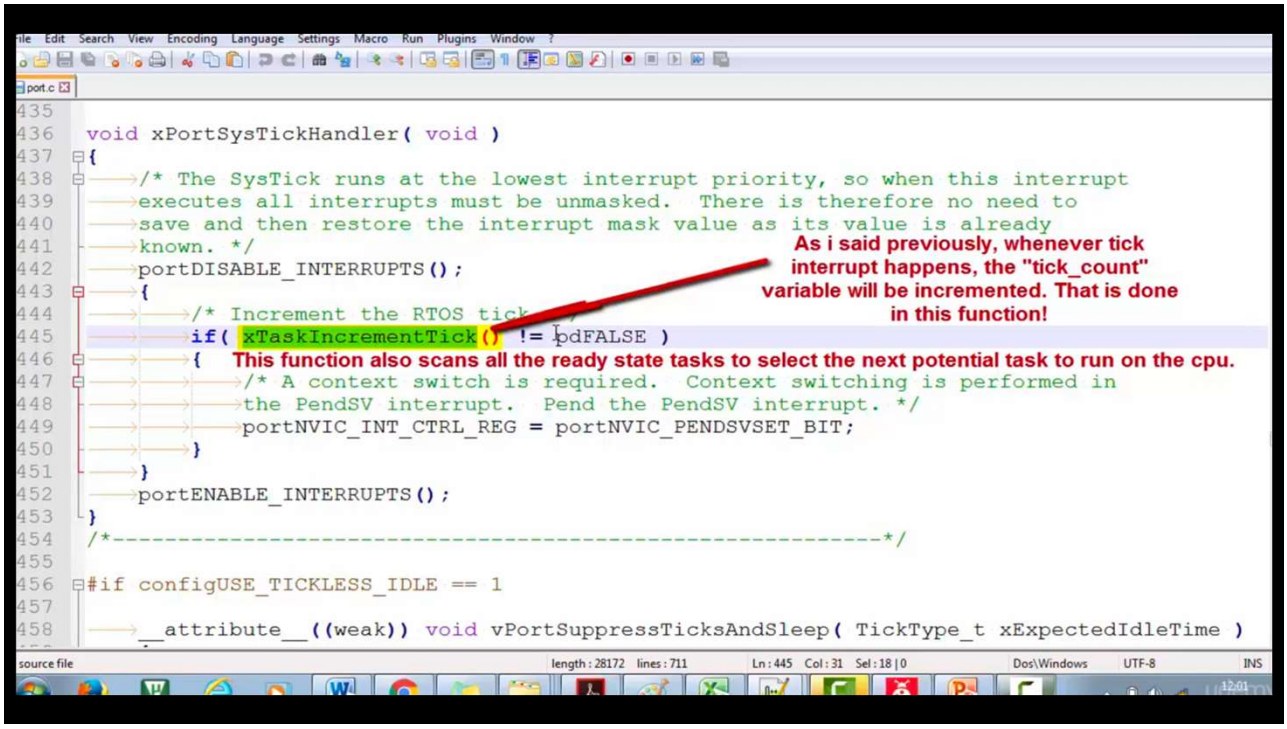
Used for Context Switching to the next potential Task



1. The tick ISR runs
2. All the ready state tasks are scanned
3. Determines which the next potential task to run
4. If found, triggers the context switching handler

So, the Tick ISR doesn't carry out Context switching , it just invokes another handler that later will carry out the context switching operation !

- Kernel will be use hardware timer peripherals to issues ticks

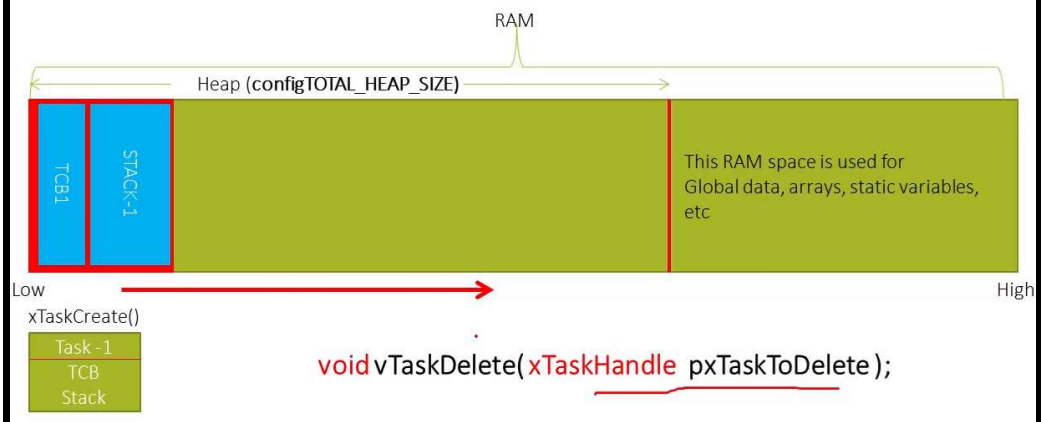


Task Implementation Function(Task Function)

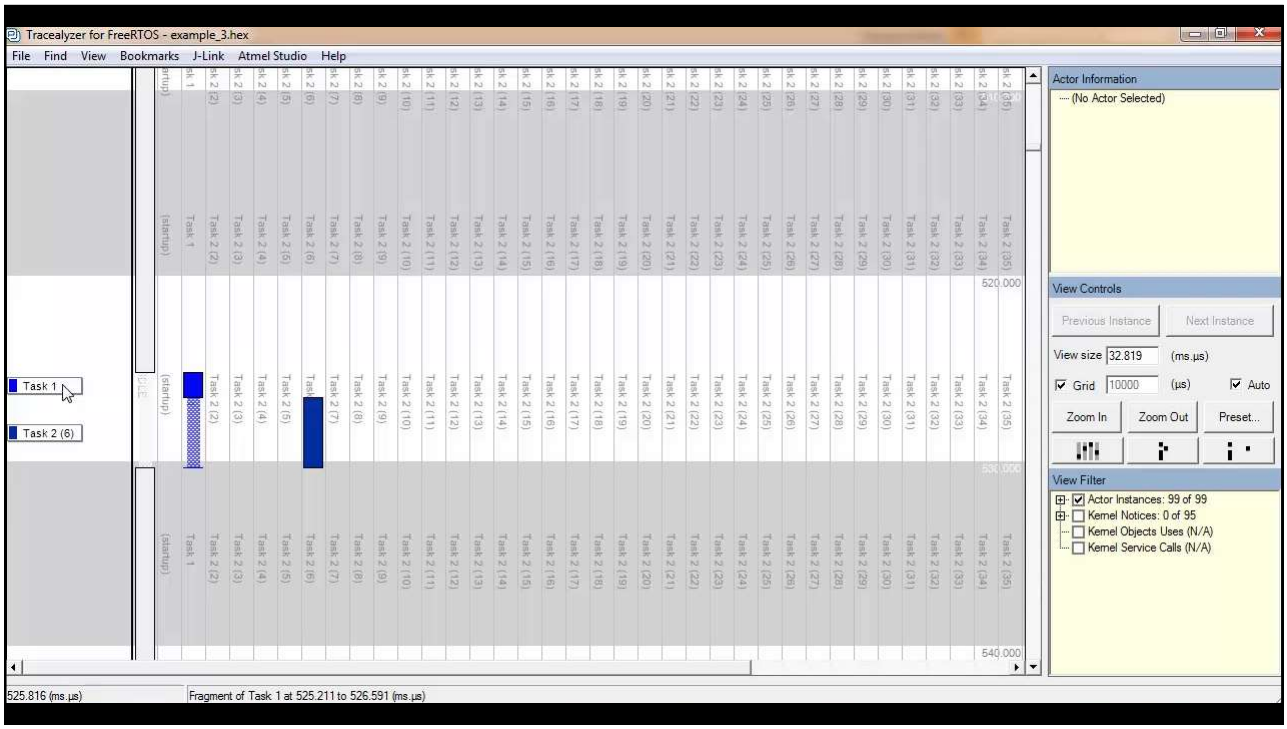
```
void ATaskFunction( void *pvParameters )
{
/* Variables can be declared just as per a normal function.  Each instance
of a task created using this function will have its own copy of the
iVariableExample variable.  This would not be true if the variable was
declared static – in which case only one 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() က task တစ်ခုဒီ function ကိုလာသုံးပြီးတာနဲ့ တန်းပြီး ဖျက်မှာ။
- ဥပမာ - task2 ခုက ဒီ function handlerကို ဘုံသုံးမယ်ဆိုရင် int နဲ့ ကြေညာထားတဲ့ iVariableExample သည် task တစ်ခုခြင်းစီအတွက် local var အဖြစ် Task ရဲ့ stack memory ထဲမှာသိမ်းမှာဖြစ်တယ်။
- Int ကို static အဖြစ်နဲ့ ကြေညာလိုက်ရင်တော့ global shared variable အဖြစ်နဲ့ task နှစ်ခုက အပြိုင်သုံးမှာဖြစ်တယ်။

Deleting a Task



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



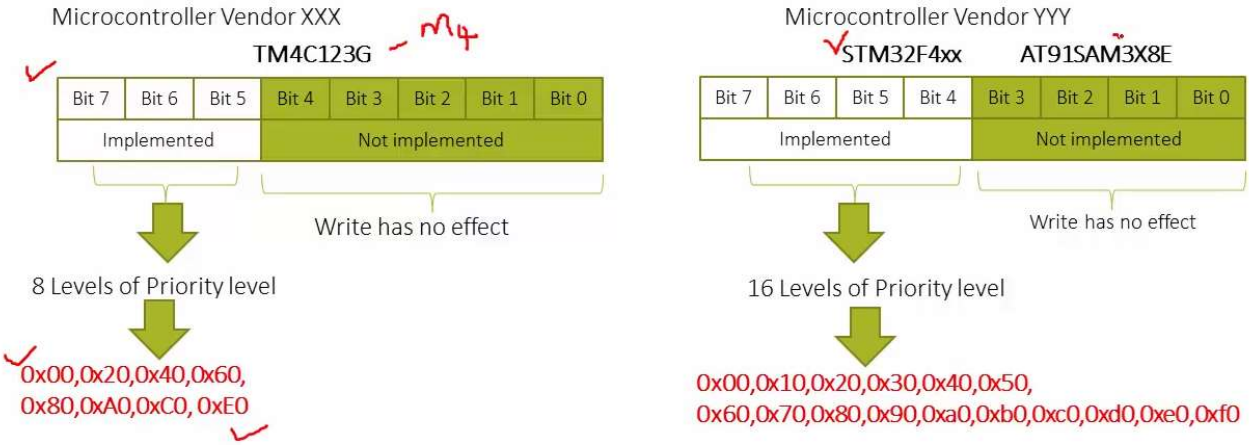
Priority register and priority levels in ARM Cortex-M Processor

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



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 ATMEL 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

TM4C123G

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

Write has no effect

8 Levels of Priority level

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

Microcontroller Vendor YYY

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

Write has no effect

16 Levels of Priority level

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

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			

Priority_register = priority_value

5

0 0 0 0 0 1 0 1 = 0

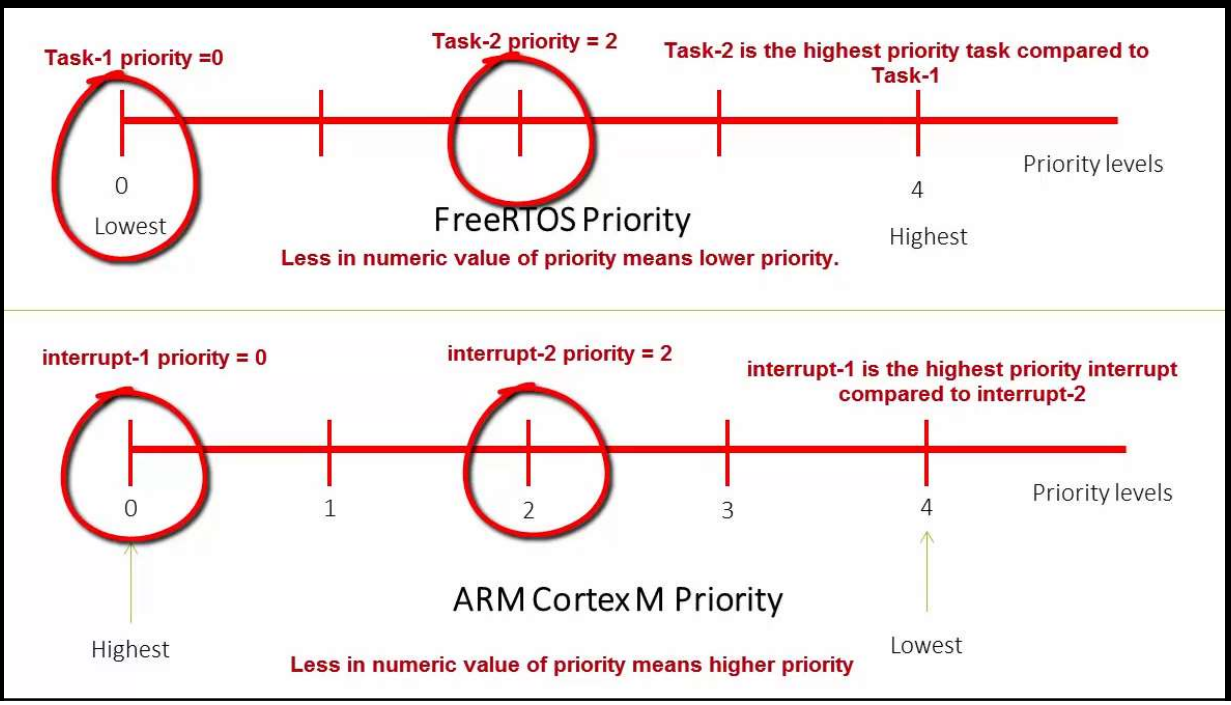
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Implemented				Not implemented			

0 1 0 1 0 0 0 0

Priority_register = priority_value << (8 - __NVIC_PRIO_BITS)

5 << (8 - 4)

FreeRTOS Task Priority Vs Processor Interrupt/Exception Priority



FreeRTOS Priority Configuration Config Items

configKERNEL_INTERRUPT_PRIORITY

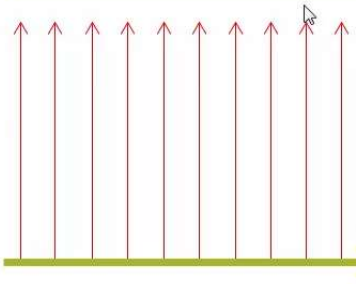
configMAX_SYSCALL_INTERRUPT_PRIORITY

configKERNEL_INTERRUPT_PRIORITY

This config item, decides the priority for the kernel Interrupts

What are the kernel interrupts ?

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



1 Systick Interrupt

Time **OS ticking**

2 PendSV interrupt

context switching

SVC interrupt

3 **Schedule the very first task**

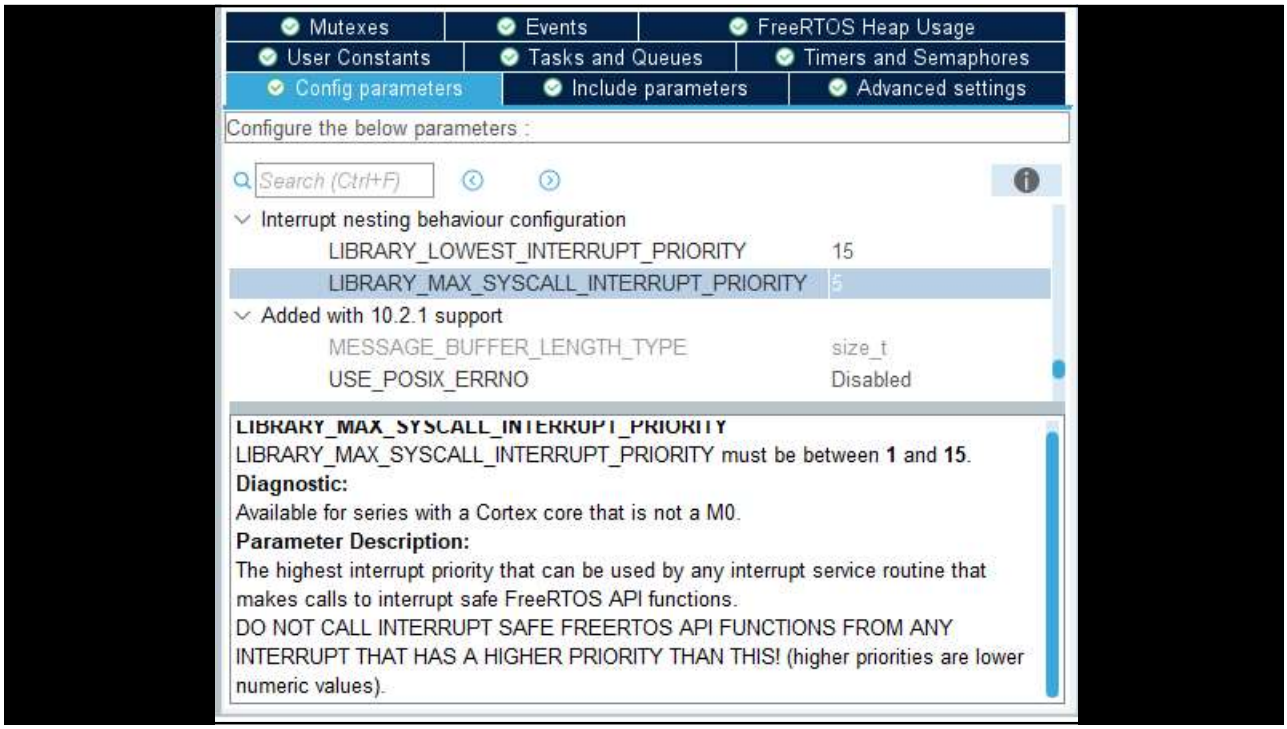
CMSIS_RTOS_F4.ioc | main.c | FreeRTOSConfig.h

```
/* The lowest interrupt priority that can be used in a call to a "set priority"
function. */
#define configLIBRARY_LOWEST_INTERRUPT_PRIORITY   15

/* 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 5

/* 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) )
/* !!!! configMAX_SYSCALL_INTERRUPT_PRIORITY must not be set to zero !!!!
See http://www.FreeRTOS.org/RTOS-Cortex-M3-M4.html. */
#define configMAX_SYSCALL_INTERRUPT_PRIORITY    ( configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY << (8 - configPRIO_BITS) )

header file. */
/* USER CODE BEGIN 1 */
#define configASSERT( x ) if ((x) == 0) {taskDISABLE_INTERRUPTS(); for( ;; );}
/* USER CODE END 1 */

/* Definitions that map the FreeRTOS port interrupt handlers to their CMSIS
standard names. */
#define vPortSVCHandler    SVC_Handler
#define xPortPendSVHandler PendSV_Handler

/* IMPORTANT: After 10.3.1 update, Systick_Handler comes from NVIC (if SYS_timebase = systick), otherwise from cmsis_os2.c */

#define USE_CUSTOM_SYSTICK_HANDLER_IMPLEMENTATION 0

/* USER CODE BEGIN Defines */
/* Section where parameter definitions can be added (for instance, to override default ones in FreeRTOS.h) */
/* 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 | sam3a4c.h

```
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
```

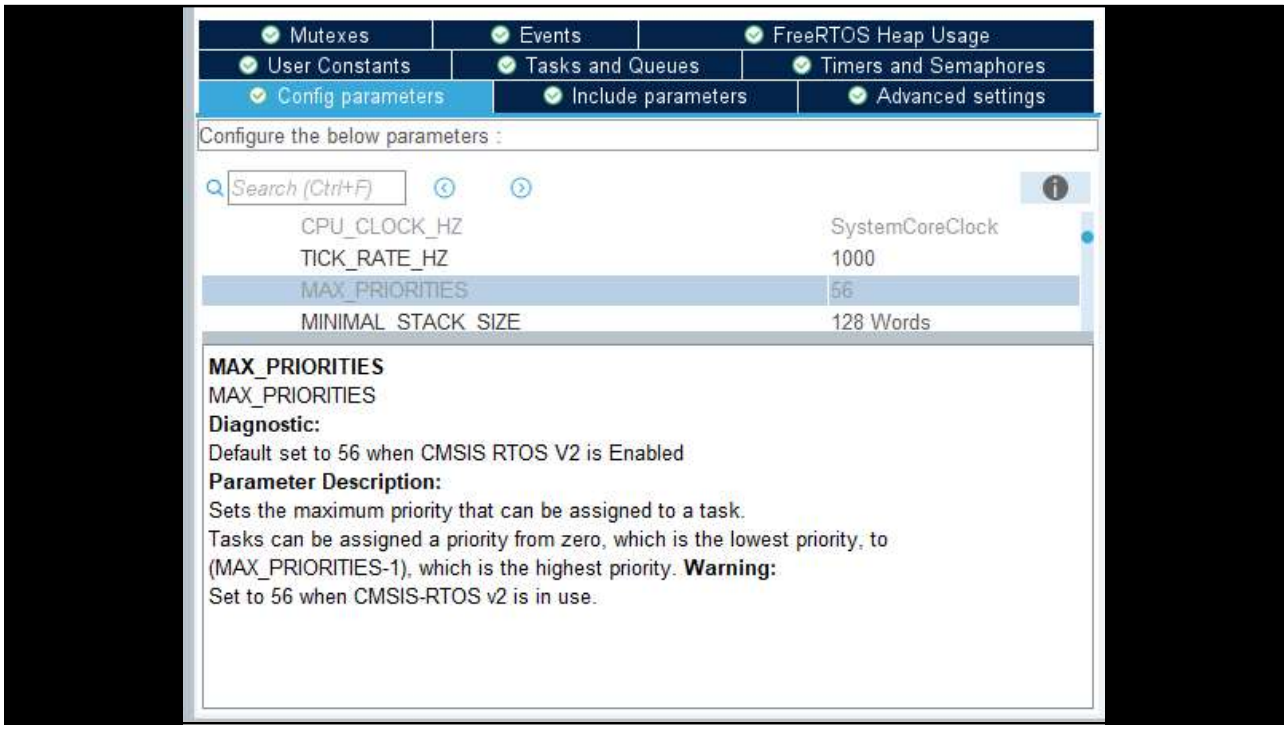


Concluding Points

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 actully 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.



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

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

```
QUEUE some_queue;

ISR_Fun(void *data)
{
   .
   Queue_write(&some_queue, data);
   next statement 1;
   next statement 2;
   .
   .
   .
}
```

```
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
}
```

Scenario of an ISR calling non-interrupt safe API

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”

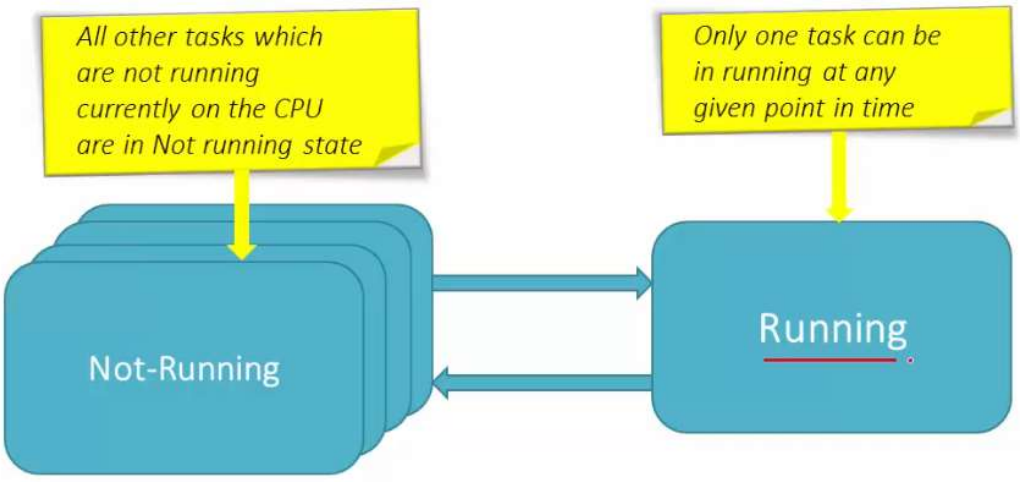
This is for the reason, Being in the interrupt Contex (i.e being in the middle of servicing the ISR) you can not return to Task Context (i.e making a task to run by pre-empting the ISR)

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

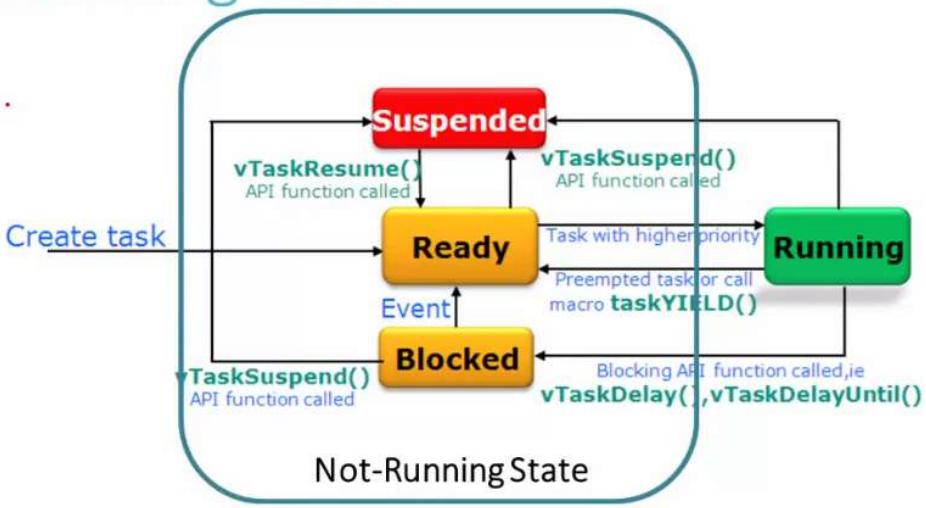
```
/* Scheduler utilities. */
#define portYIELD()                                                    \
{                                                                      \
    /* Set a PendSV to request a context switch. */                    \
    portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;                    \
                                                                       \
    /* Barriers are normally not required but do ensure the code is completely  \
    within the specified behaviour for the architecture. */            \
    __asm volatile( "dsb" ::: "memory" );                              \
    __asm volatile( "isb" );                                           \
}
```

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



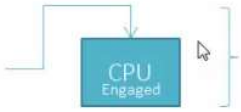
What is blocking of a Task ?

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.

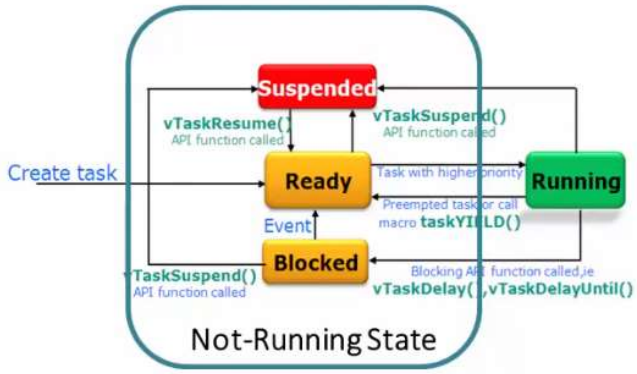
2. For Synchronization –For example, a task may enter the Blocked state to wait for data to arrive on a queue. When the another task or interrupt fills up the queue , the blocked task will be unblocked.

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

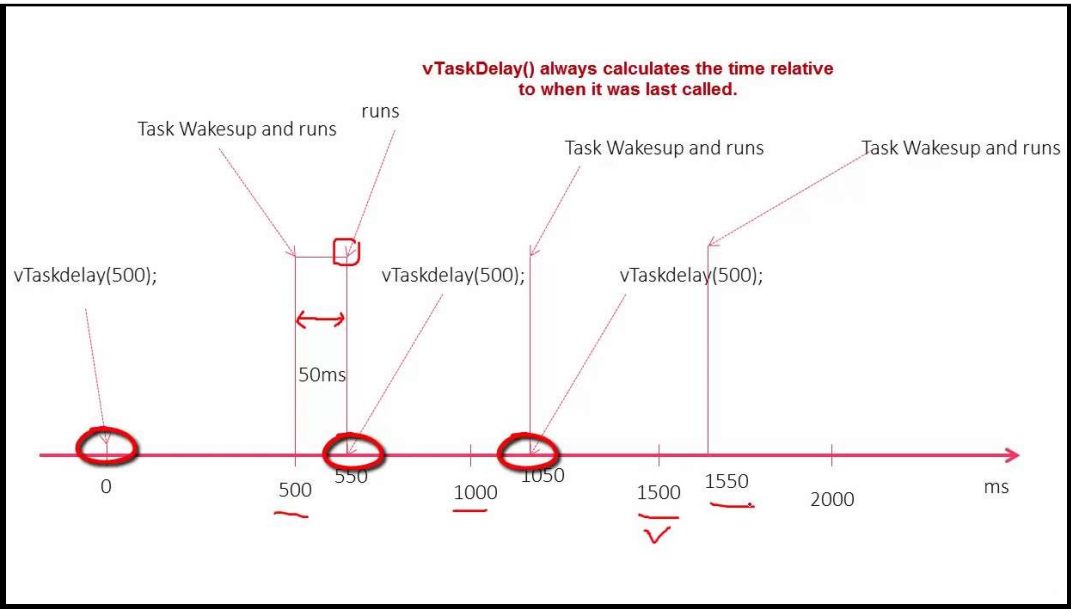
At any given point, only one task can be in the running state, and other tasks will be in not running state

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.



- 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**

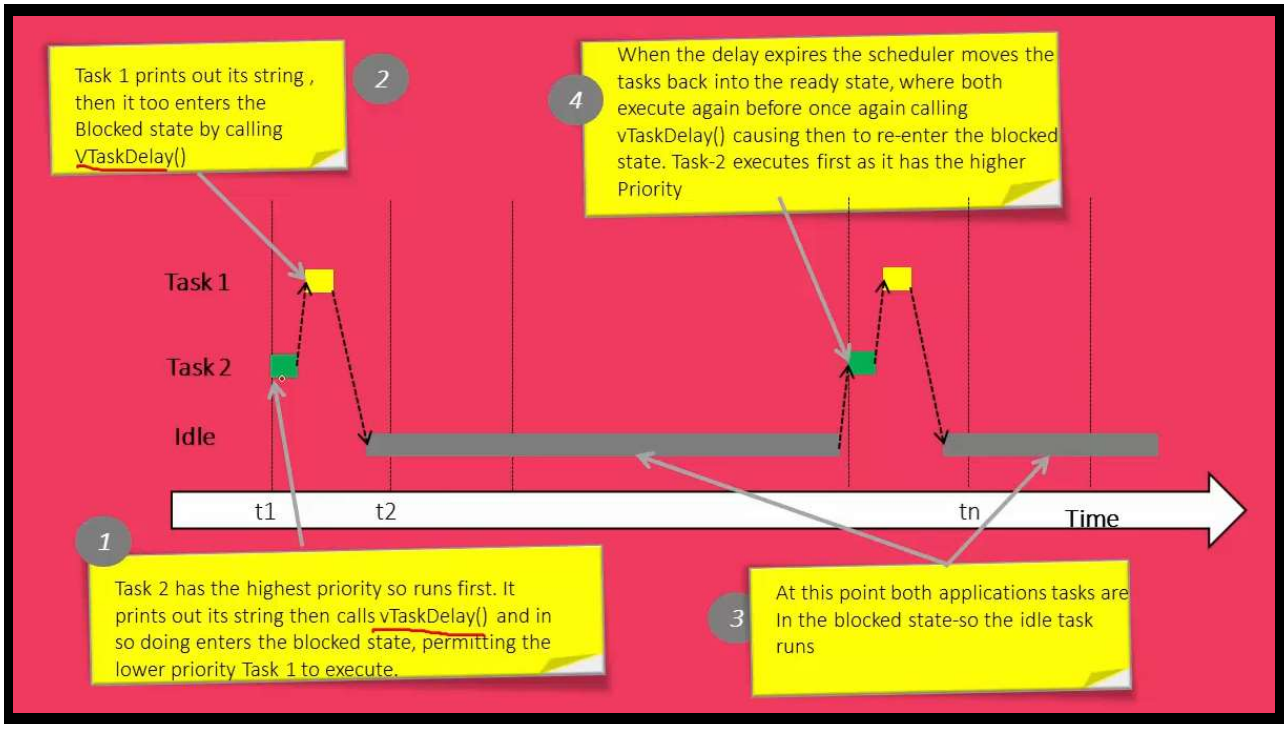
vTaskDelayUntil() specifies an ***absolute*** **time at which the task wishes to unblock.[**Time is calculated relative to the last wake up time of the task **]**

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

- ဒီ api func လေးကတော့ PreviousWakeTime ကို ပြန်ယူထားပီး လိုအပ်တဲ့ အချိန်ကို ပြန်ပီးတိုးပေးမှာဖြစ်တဲ့အတွက် task တစ်ခုအတွက် delay အတိအကျကိုရရှိနိုင်မှာဖြစ်ပါတယ်။

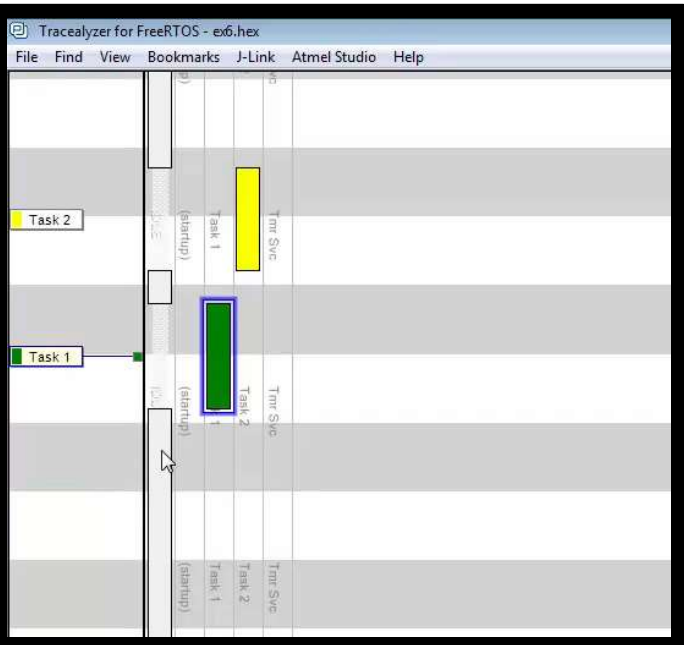
Convert time to ticks

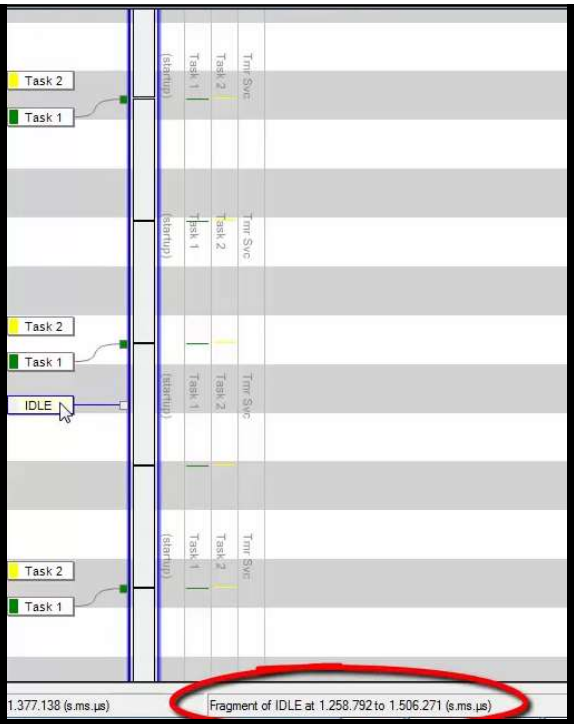
- Kernel က time ကိုနားမလည်တဲ့အတွက် ထည့်လိုက်တဲ့ time ကို tick ပြန်တွက်ပါတယ်။
- configTICK_RATE_HZ = 1000 Hz သည် kernel အတွက်ဖြစ်ပါတယ်။ 1000 Tick interrupt မှာ 1 sec နဲ့ညီမျှပါတယ်။
- TICK_RATE_MS = 1/1000 = 1ms
- အဲ့အတွက် 2 ticks အင်တာရပ် နှစ်ခုကြားမှာ 1 ms ကြာမှာပါ။
- အုပမာ - 250 Hz ကို သတ်မှတ်ထားမယ်ဆွင် 1/250 = 4 ms ကြာမှာပါ။



အရှေ့ကပုံရဲ့ရှင်းပြချက်

- Task 2 က 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။





Free RTOS HOOK

- Idle HOOK (or callback)
- Tick Hook

Idle Task

In your freeRTOS Application , if you delete any dynamically created kernel services like say semaphores, queues or tasks, then freeing of memory or cleaning up of those deleted services will be done inside the idle task.

So, idle task is the one which handles the freeing of memory of the deleted service

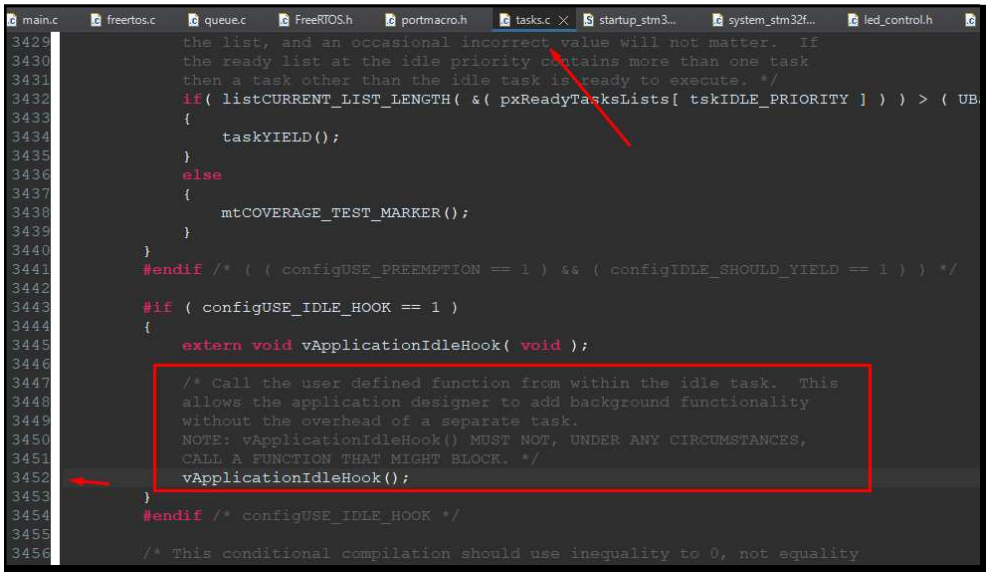
Idle Task

It is therefore important in applications that whenever you delete something using freeRTOS APIs, then you must also make sure that idle task is not starved of processing time.

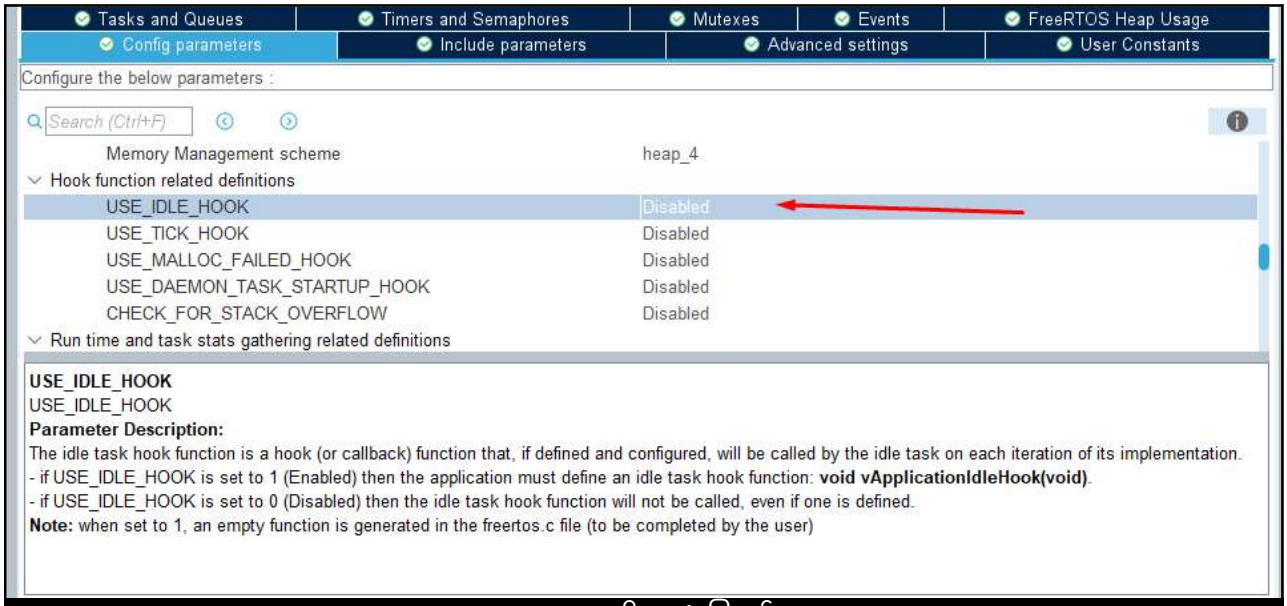
That means you must allow idle task to run , otherwise the memory occupied by those deleted kernel services will be active unnecessarily .

```
void vTaskStartScheduler( void )
{
BaseType_t xReturn;

    /* Add the idle task at the lowest priority. */
    #if( configSUPPORT_STATIC_ALLOCATION == 1 )
    {
        StaticTask_t *pxIdleTaskTCBBuffer = NULL;
        StackType_t *pxIdleTaskStackBuffer = NULL;
        uint32_t ulIdleTaskStackSize;

        /* The Idle task is created using user provided RAM - obtain the
        address of the RAM then create the idle task. */
        vApplicationGetIdleTaskMemory( &pxIdleTaskTCBBuffer, &pxIdleTaskStackBuffer, &ulIdleTaskStackS
        xIdleTaskHandle = xTaskCreateStatic(    prvIdleTask,
                                                configIDLE_TASK_NAME,
                                                ulIdleTaskStackSize,
                                                ( void * ) NULL, /*lint !e961.  The cast is not redund
                                                portPRIVILEGE_BIT, /* In effect ( tskIDLE_PRIORITY | p
                                                pxIdleTaskStackBuffer,
                                                pxIdleTaskTCBBuffer ); /*lint !e961 MISRA exception, j

        if( xIdleTaskHandle != NULL )
        {
            xReturn = pdPASS;
        }
        else
        {
```

- Scheduler ကို start လုပ်တာနဲ့ တပြိုင်တည်း priority zero ဖြစ် IdleTask ကို တည်ဆောက်ထားပုံ



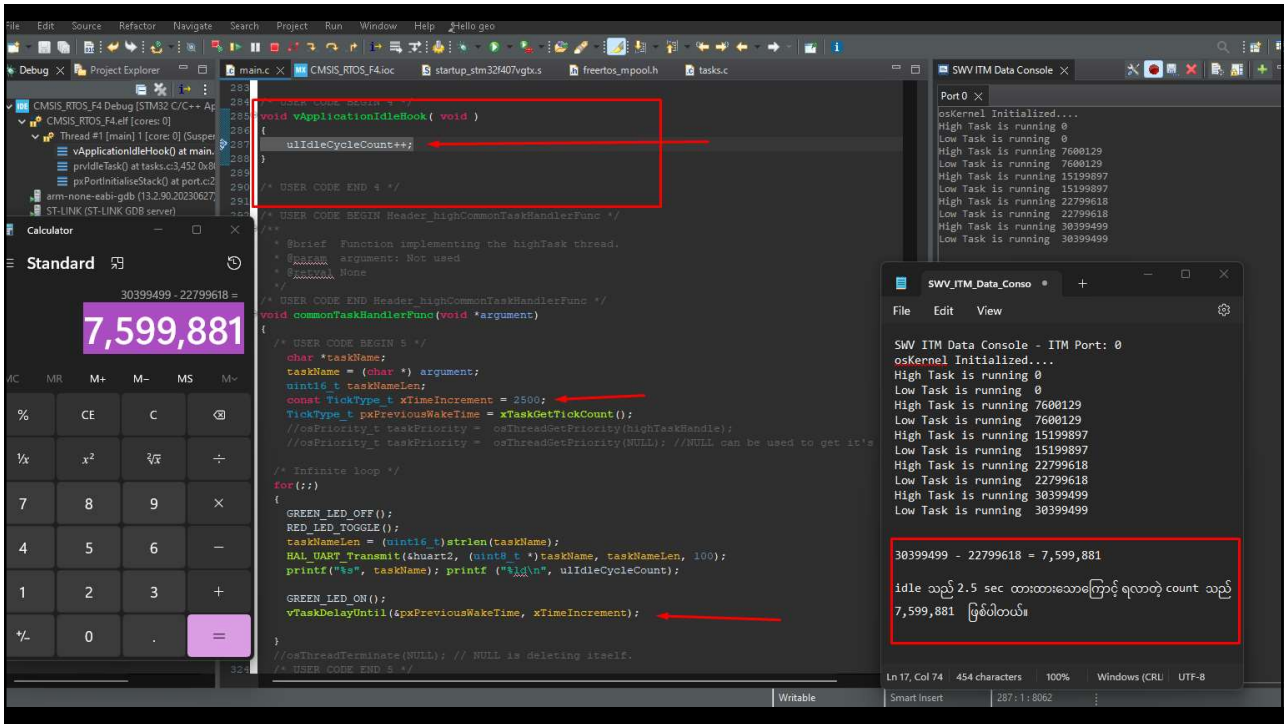
- Idle state CPU ကဘာအလုပ်မှမလုပ်ပေမယ့် vApplicationIdleHook ကိုအသုံးပြုပြီး
- ကိုယ်လုပ်ချင်တဲ့ကိစ္စရပ်များကိုလုပ်ဆောင်နိုင်။ idleHook function လို့ခေါ်သည်။



- Idle HOOK (or callback) function ကိုအသုံးပြုရန် cubeMX
- Configuration ဖစ်သည်။

Idle task hook function

- Idle task မှာ callback ပြန်ခေါ်မယ့် function ဖြစ်တယ်။
- void vApplicationIdleHook(void);
- Idle task run တဲ့အချိန်တိုင်းမှာ hook function ကိုခေါ်၍ အသုံးဝင်တဲ့ကိစ္စရပ်များကိုလုပ်ဆောင်နိုင်ပါတယ်။
- E.g Mcu ကို low power mode လုပ်ဆောင်ပေးတာ။ မသုံးတော့တဲ့ kernel objects တွေကို stack, heap တို့ကနေ deallocate လုပ်ပေးတာမျိုးတို့ဖြစ်ပါတယ်။



Low Power Mode For MCU

```
static void prvEnterSleepModeUsingWFI(void)
{
    PMC->PMC_FSMR &= (uint32_t)~PMC_FSMR_LPM;
    SCB->SCR &= (uint32_t)~SCR_SLEEPDEEP;
    __WFI();
}
```

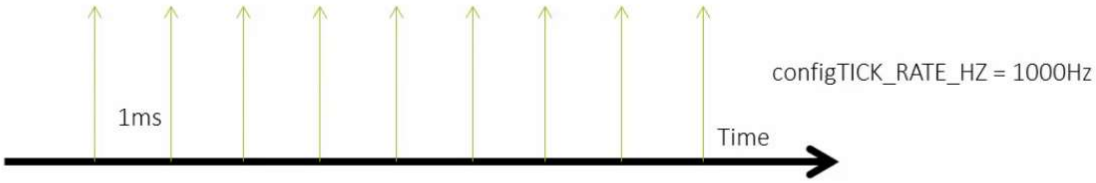
- WFI() Wait for Interrupts

```
static void prvEnterSleepModeUsingWFE(void)
{
    PMC->PMC_FSMR &= (uint32_t)~PMC_FSMR_LPM;
    SCB->SCR &= (uint32_t)~SCR_SLEEPDEEP;
    __WFE();
}
```

- WFE() Wait For Events

```
void vApplicationIdleHook( void )
{
    /* This hook function does nothing but increment a counter. */
    ulIdleCycleCount++;
#if 1
    /* lets enter sleep mode here, CPU clock will be off */
    pmc_sleep(SAM_PM_SMODE_SLEEP_WFI);
    //prvEnterSleepModeUsingWFI();
#endif
```

Tick hook function



The tick interrupt can optionally call an application defined hook (or callback) function - the tick hook

You can use tick hook function to implement timer functionality.

The tick hook will only get called if configUSE_TICK_HOOK is set to 1 within FreeRTOSConfig.h

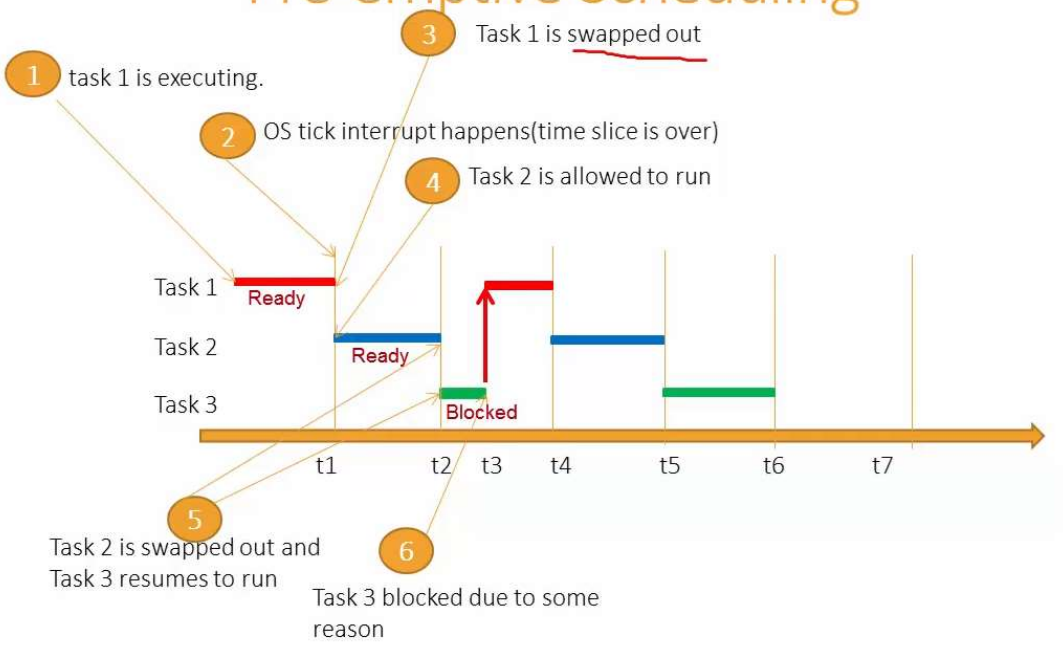
Implement this function : void vApplicationTickHook(void); in your application

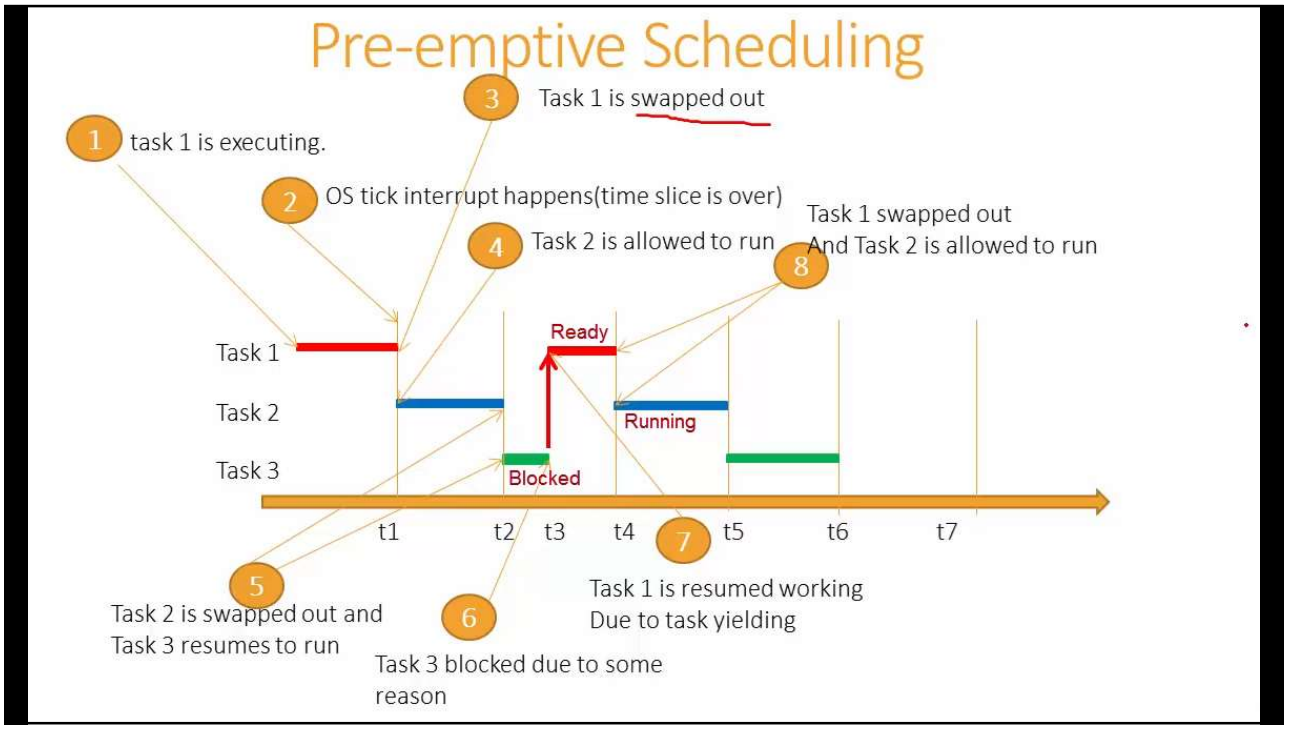
vApplicationTickHook() executes from within an ISR so must be very short,

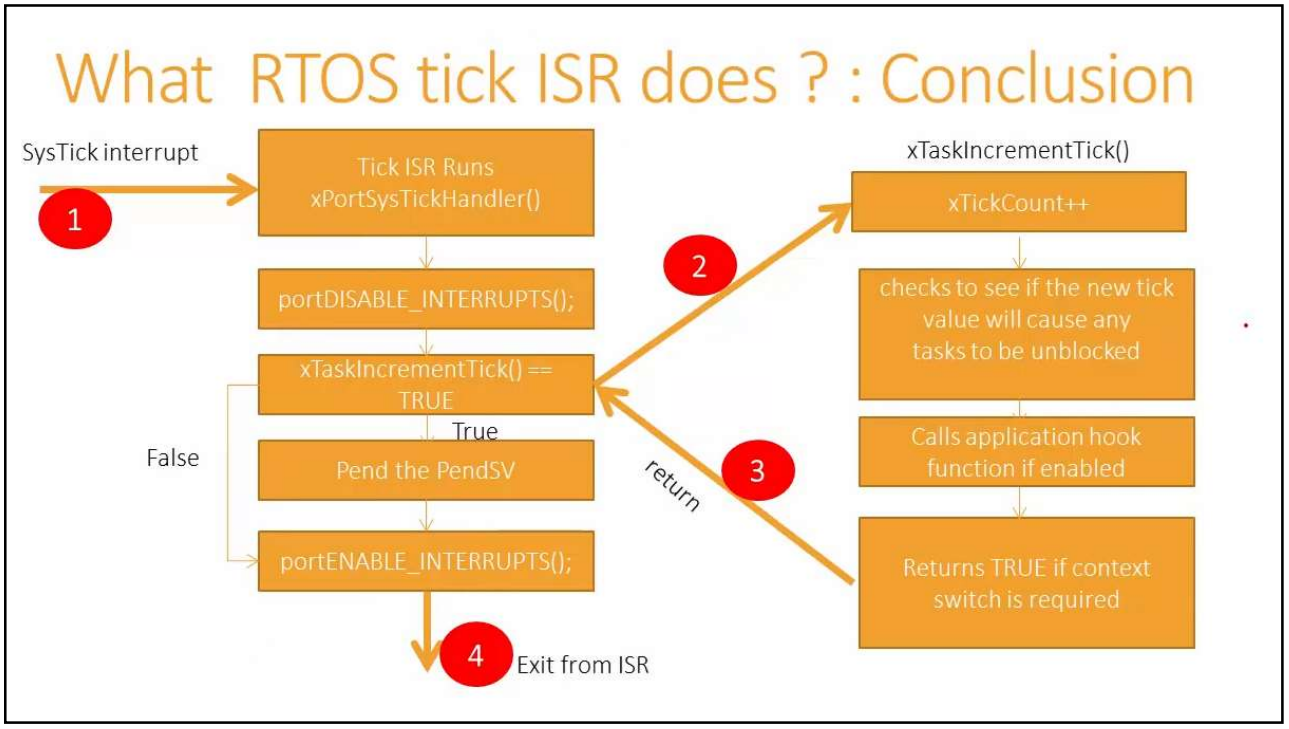
Preemption

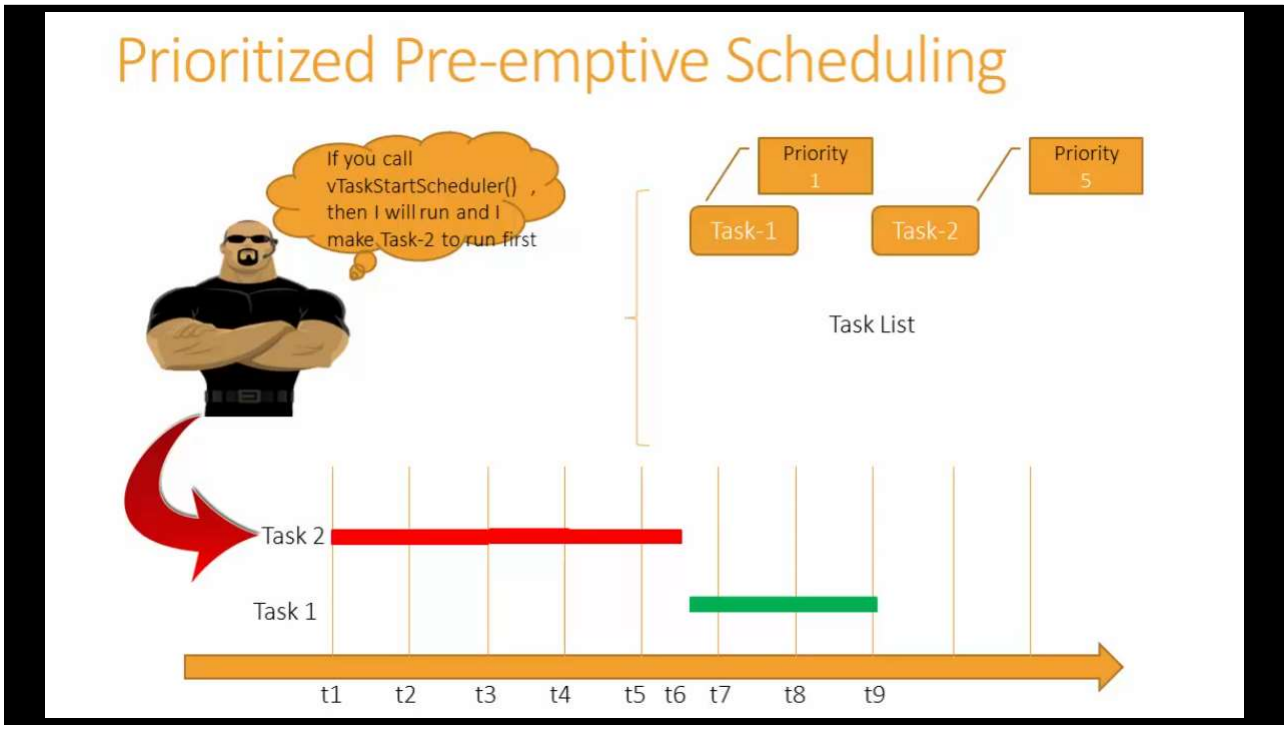
Preemption is the act of temporarily interrupting an already executing task with the intention of removing it from the running state without its co-operation .

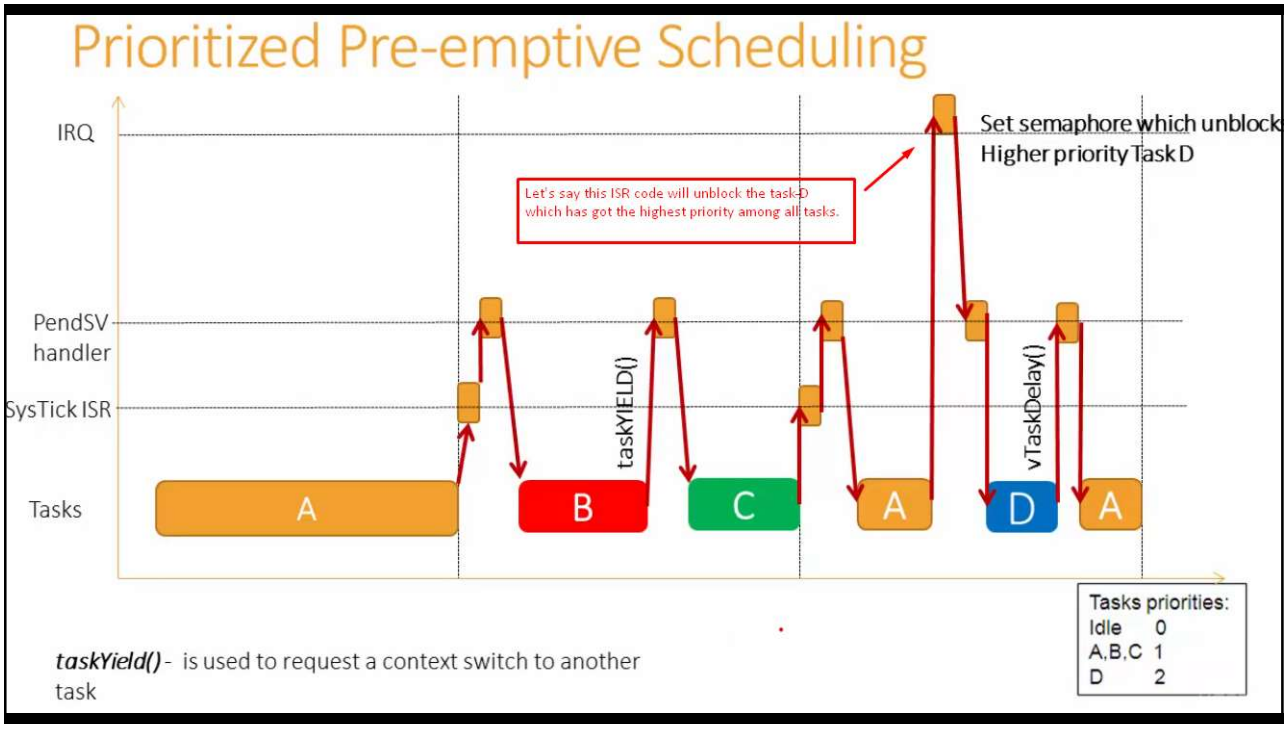
Pre-emptive Scheduling



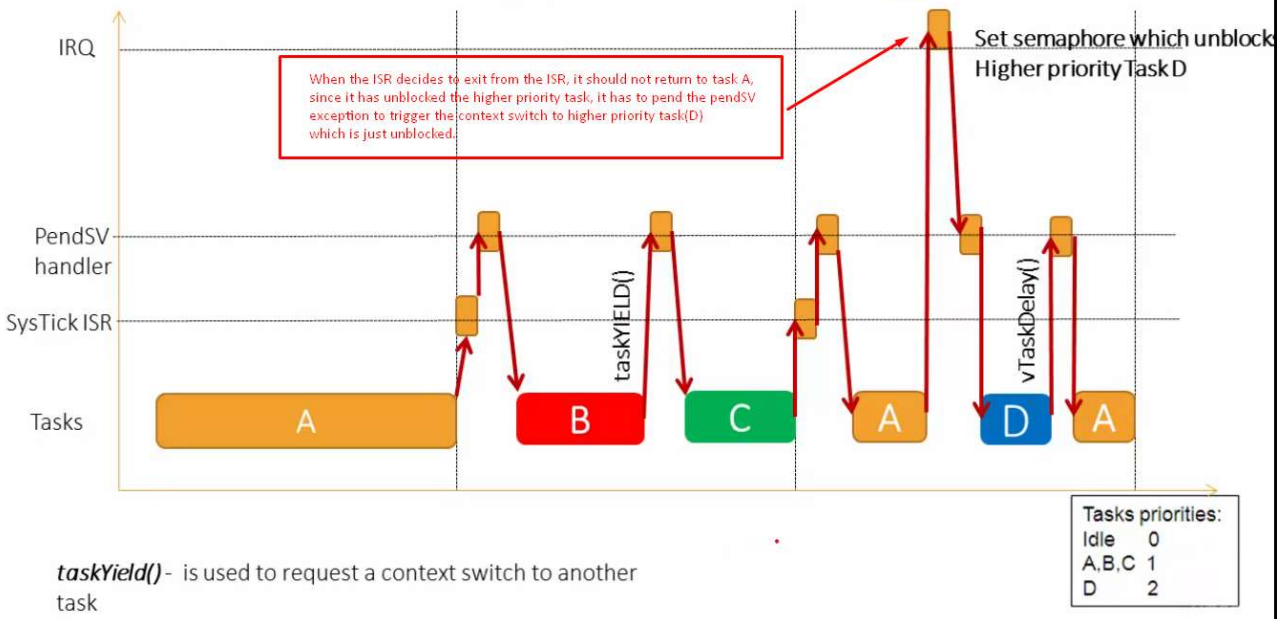

Pre-emptive Scheduling
1 task 1 is executing.
2 OS tick interrupt happens(time slice is over)
3 Task 1 is swapped out
4 Task 2 is allowed to run
5 Task 2 is swapped out and Task 3 resumes to run
6 Task 3 blocked due to some reason
7 Task 1 is resumed working Due to task yielding
8 Task 1 swapped out And Task 2 is allowed to run
Task 1
Task 2
Task 3
Ready
Running
Blocked
t1
t2
t3
t4
t5
t6
t7


What RTOS tick ISR does ? : Conclusion
SysTick interrupt
1
Tick ISR Runs xPortSysTickHandler()
portDISABLE_INTERRUPTS();
xTaskIncrementTick() == TRUE
True
False
Pend the PendSV
portENABLE_INTERRUPTS();
4 Exit from ISR
2
xTaskIncrementTick()
xTickCount++
checks to see if the new tick value will cause any tasks to be unblocked
Calls application hook function if enabled
Returns TRUE if context switch is required
return
3


Prioritized Pre-emptive Scheduling
If you call vTaskStartScheduler() , then I will run and I make Task-2 to run first
Priority 1
Priority 5
Task-1
Task-2
Task List
Task 2
Task 1
t1
t2
t3
t4
t5
t6
t7
t8
t9


Prioritized Pre-emptive Scheduling
IRQ
PendSV handler
SysTick ISR
Tasks
Let's say this ISR code will unblock the task-D which has got the highest priority among all tasks.
Set semaphore which unblock Higher priority Task D
taskYIELD()
vTaskDelay()
A
B
C
A
D
A
Tasks priorities:
Idle 0
A,B,C 1
D 2
taskYield() - is used to request a context switch to another task

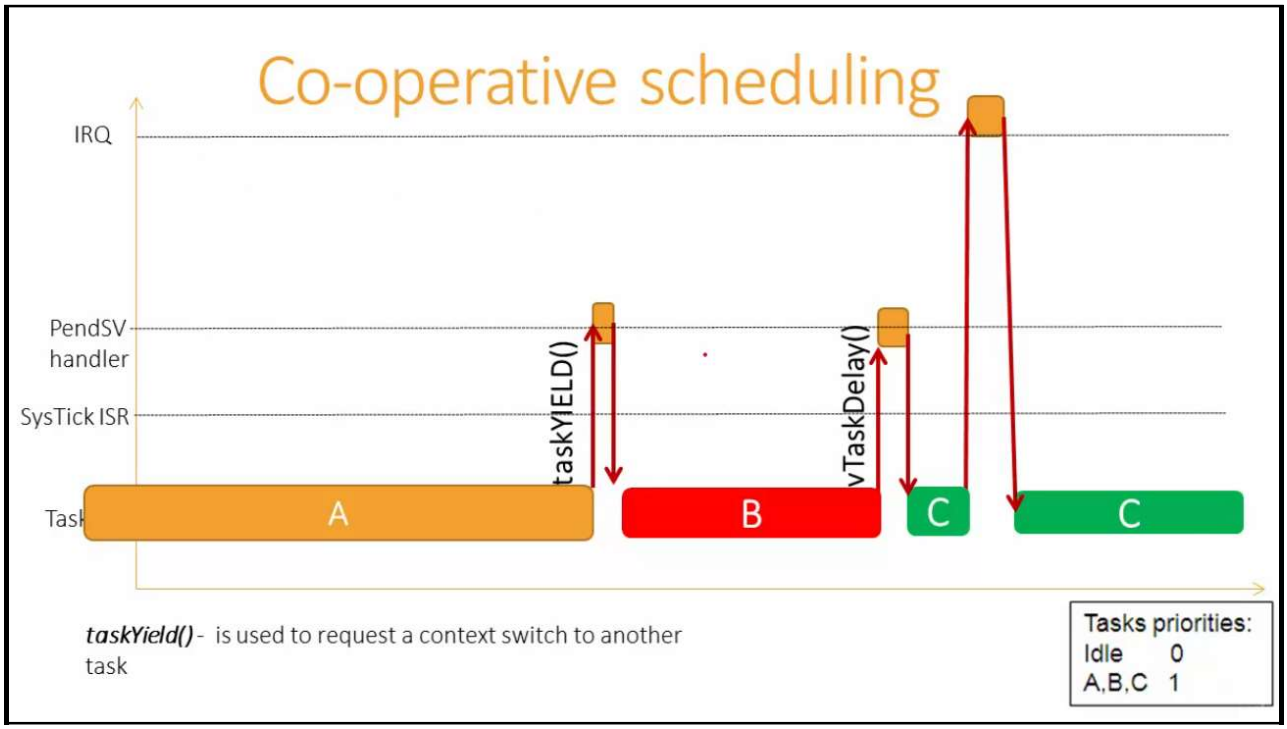
Prioritized Pre-emptive Scheduling

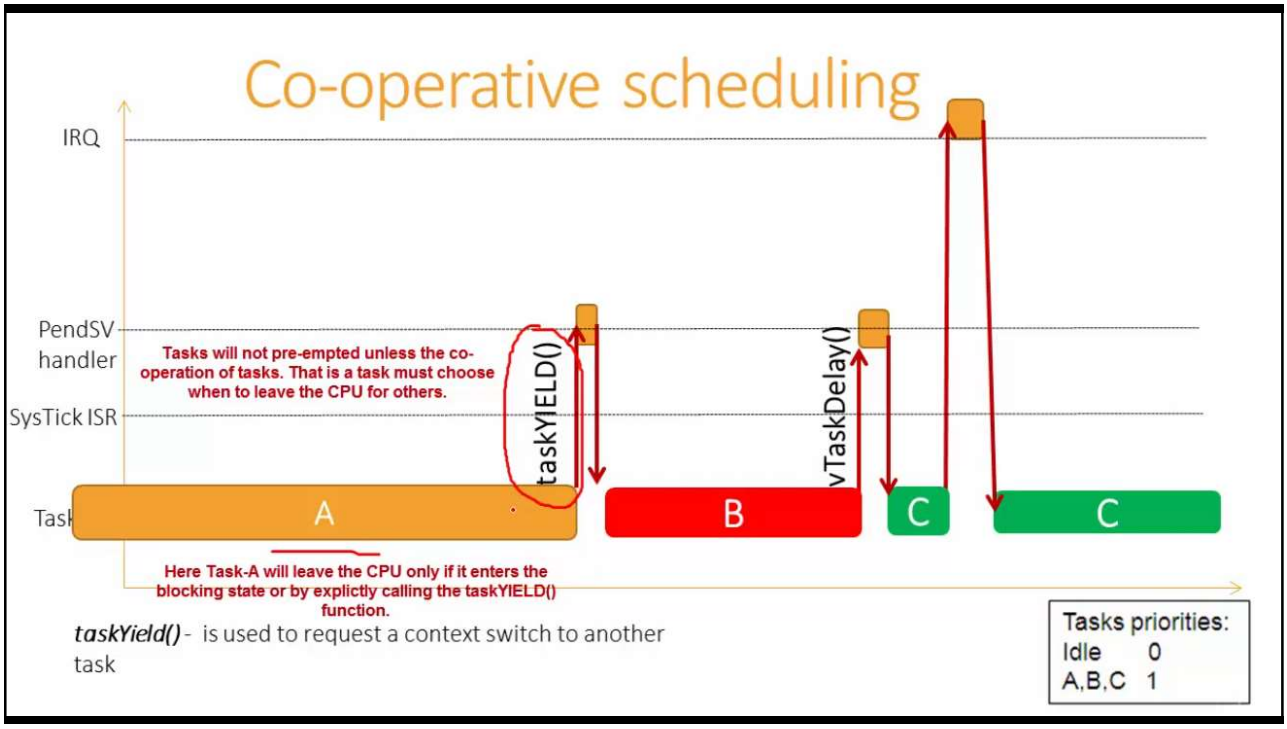


taskYield() - is used to request a context switch to another task

Co-Operative scheduling

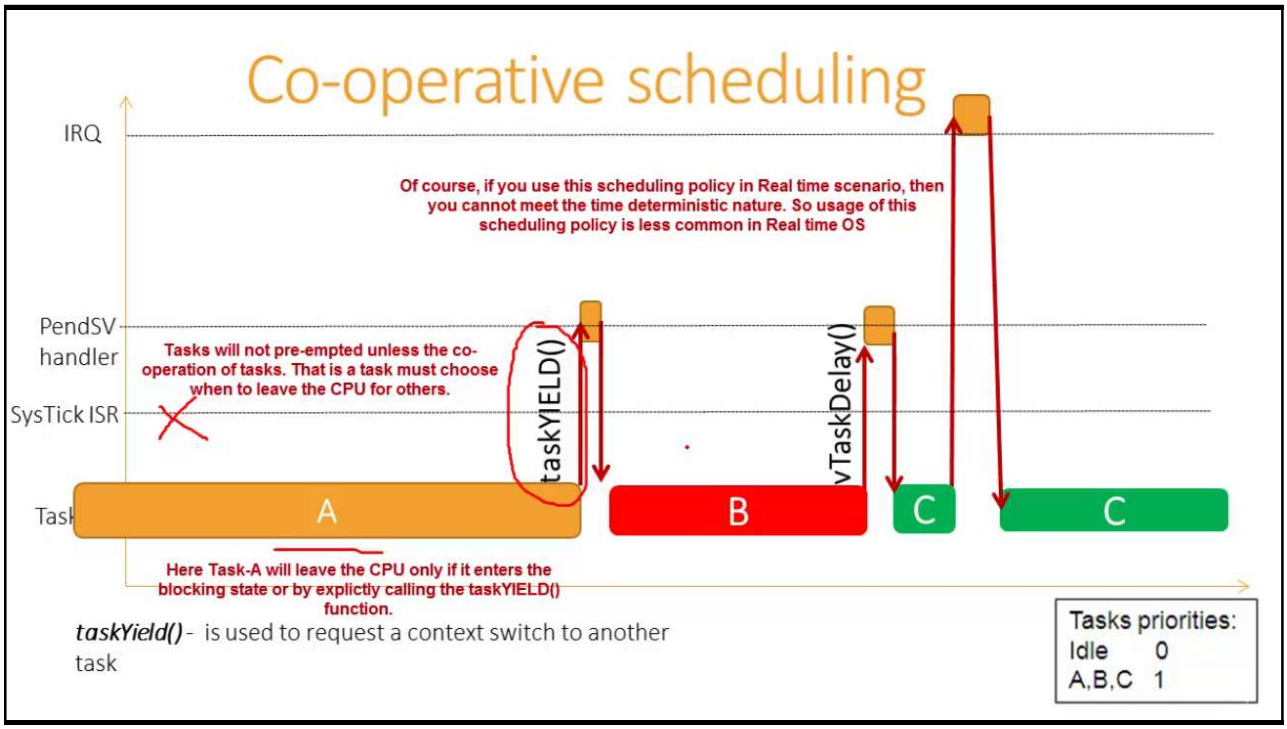
As the name indicates, it is a co-operation based scheduling . That is Co-operation among tasks to run on the CPU.


Co-operative scheduling
IRQ
PendSV handler
SysTick ISR
Task
taskYIELD()
vTaskDelay()
A
B
C
C
taskYield() - is used to request a context switch to another task
Tasks priorities:
Idle 0
A,B,C 1


Co-operative scheduling
IRQ
PendSV handler
SysTick ISR
Task
Tasks will not pre-empted unless the co-operation of tasks. That is a task must choose when to leave the CPU for others.
taskYIELD()
vTaskDelay()
A
B
C
C
Here Task-A will leave the CPU only if it enters the blocking state or by explictly calling the taskYIELD() function.
taskYield() - is used to request a context switch to another task
Tasks priorities:
Idle 0
A,B,C 1

- Co-operative scheduling မှာ context switching သည်၊ taskYield() ကို ခေါ်သုံးထားတဲ့အချိန်မျိုးမှာနဲ့ task တစ်ခုကို ဘယ်အချိန်မှာ block လုပ်ပီး ဘယ်အချိန်မှာ တခြား task အတွက် unblock လုပ်ပေးရမယ့်အချိန်မျိုးကို သိမှ pendSV Handler က task swapping လုပ်ပေးမှာဖြစ်ပါတယ်။
- SysTick ISR က useless ဖြစ်သွားပါတယ်။
- Tasks တွေအားလုံးဟာ ဘယ်တော့မှ Kernel Sys Tick ကိုသုံးပီးဘယ်တော့မှ context switching လုပ်ပေးမယ့် pendSV Handler ကိုခေါ်သုံးမှာမဟုတ်ဘူး။ taskYield() တစ်ခုကိုပဲအားကိုးပီး switching လုပ်ကြရမှာပါ။

- အာ့ကြောင့် real time application တွေအတွက် ဒီ scheduling policy က အလုပ်ဖြစ်မှာမဟုတ်ပါ။
- ဒါပေမယ့် freeRTOS မှာ co-operative scheduling policy ကို အသုံးပြုခွင့်ပေးထားပါတယ်။



PART one Finished ...

TBC - > PART two