# COMPLETE GUIDE TO RTOS (REAL-TIME OPERATING SYSTEMS) WITH FREERTOS

## RTOS INTRODUCTION

### What is a Real Time Application (RTA)?

Myth: Real Time Computing is equivalent to very fast computing ✗

Truth: Real time computing is equivalent to predictable computing ✓

### What is Real-Time?

Real time means that events are executed with precise guarantee within a specific deadline not with pure raw speed.

Having more processors, more RAM, faster bus interfaces does NOT make a system Real-Time

A Real-Time system is one which the correctness of the computations not only depend upon the logical correctness of the computation, but also upon the time at which the result is produced. If the timing constraints are not met, a system failure is said to have occured.


NON-Real time Application

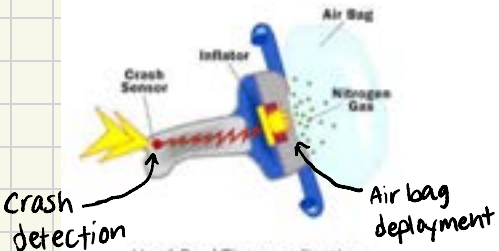
Real time Application

For example, we can see from the above graph that the Blue RTA line is much more steady. Thats because RTA's must have predictable and guaranteed timing and response time. the NRTA line shows the response time for a non-RTA, like a computer, Although sometimes the execution time can be higher, the main goal for NRTA is to have fastest response, RTA instead has main goal to have predictable response as per the requirement deadlines.

## What are Real Time Applications?

- RTA's are not fast-executing Applications.
- RTA's are time deterministic applications, this mean that their response time to events is constant, regardless of other conditions
- There could be minor deviations in an RTA response time, in the scale of a few ms or seconds. These kinds of RTA are called soft real time applications

- A hard real time application must complete within a given time limit. Failure to do so will result in absolute failure of the system.



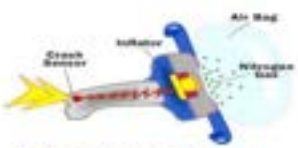
Hard-Real Time application


May be soft-real time application ?
Delay is tolerable

More examples


Missile guidance and control systems


Ani-Lock Breaking System


Airbag Deployment



Stock Market Website

| Industry | Hard Real-time | Soft Real-time |
|---|---|---|
| Aerospace | Fault Detection, Aircraft Control | Display Screen |
| Finance | ATMs | Stock Market Websites |
| Industrial | Robotics, DSP | Temperature Monitoring |
| Medical | CT Scan, MRI, fMRI | Blood Extraction, Surgical Assist |
| Communications | QoS | Audio/Video Streaming, Networking, Camcorders |

# What is an RTOS (Real-time Operating System)?

It's an operating system specially designed to run applications with very precise timing and a high degree of reliability.

To be considered as "real-time", an operating system must have a known maximum time for each of the critical operations that it performs. Some of these operations include:

- Handling of interrupts and internal system exceptions
- Handling of critical sections of code
- Scheduling Mechanisms, etc.

## RTOS vs GPOS (General-Purpose Operating Systems)


## RTOS vs GPOS: Task Scheduling

### GPOS Task Scheduling:

A GPOS is programmed to handle scheduling in such a way that it manages to achieve high throughput.

Throughput means - The total number of processes that complete their execution per unit time

Sometimes, execution of a high priority process will get delayed in order to serve 5 or 6 low priority tasks. High throughput is achieved by serving 5 low priority tasks than by serving a single high priority one.

In a GPOS, the scheduler typically uses a fairness policy to dispatch threads and processes onto the CPU.

Such a policy enables the high overall throughput required by desktop and server applications, but offers no guarantee that high-priority, time critical threads or processes will execute in preference to lower-priority threads.

## RTOS task scheduling:

On the other hand, in RTOS, threads execute in the order of their priority. If a high-priority thread becomes ready to run, it will take over the CPU from any lower-priority thread that may be executing.

Here, a high priority thread gets executed over the low priority ones. All "low priority thread execution" will get paused. A high priority thread execution will get overidden only if a request comes from an even higher priority thread.

## Does that mean RTOS's have poor throughput?

An RTOS may yield less throughput than the General Purpose Operating systems because an RTOS always favors the higher priority tasks to execute first, but this does not mean that RTOS's have poor throughput.

A quality RTOS will still deliver decent overall throughput but can sacrifice throughput for being deterministic or to achieve time predictability.

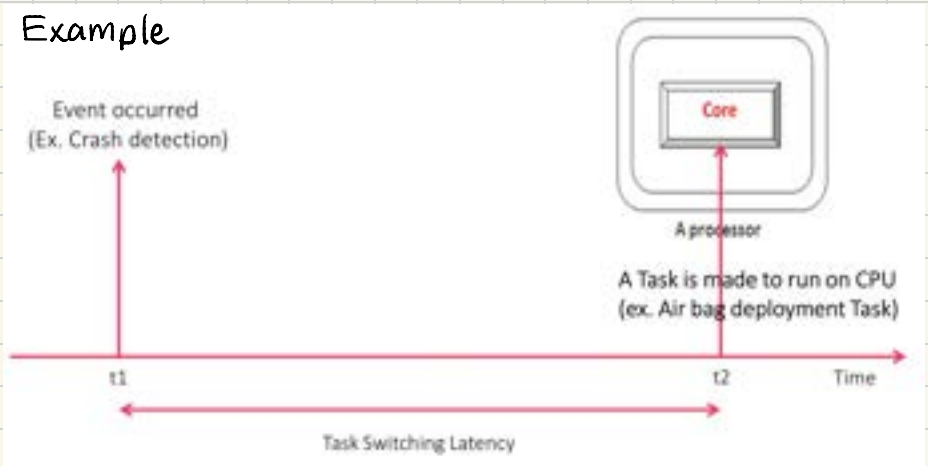
| GPOS | RTOS |
| --- | --- |
| Meet Higher throughput | Meet time predictability |

For an RTOS, achieving predictability or time deterministic nature is more important than throughput, but for the GPOS, achieving a higher throughput for user convenience is more important.

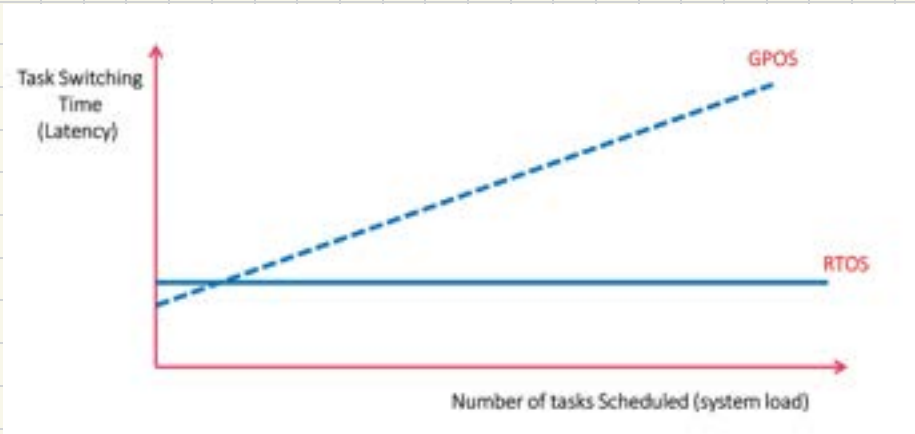
## RTOS vs GPOS: Task Switching Latency

In computing, Latency means: The time that elapses between a stimulus and the response to it.

Task switching Latency means that time gap between the Triggering of an event and the time at which the task that takes care of that event is allowed to run on the CPU.



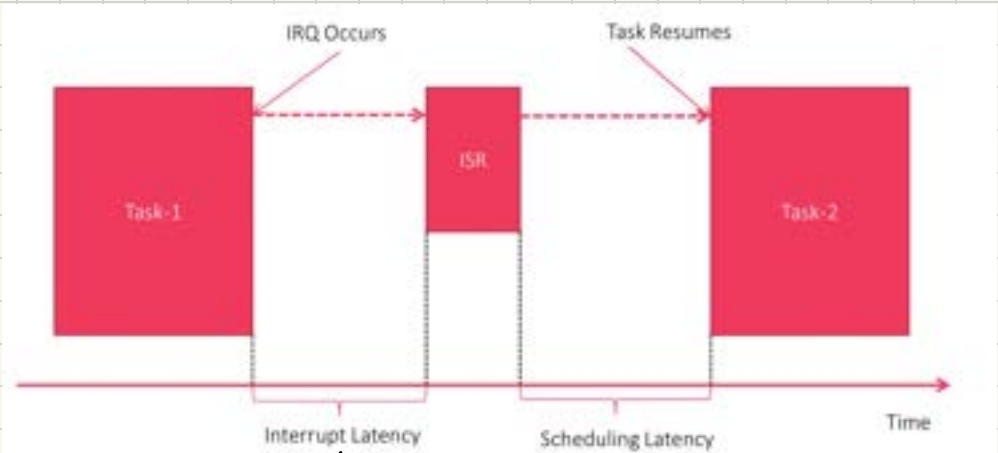

GPOS: Task Switching Latency may vary significantly
RTOS: Task Switching Latency remains almost constant




## RTOS vs GPOS: Interrupt Latency

IRQ: Interrupt Request

ISR: Interrupt Service Routine




An RTOS should have this latency as minimal as possible and must be deterministic

Time taken for the context switching operation. This latency for an RTOS must also be as minimal as possible

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

## RTOS vs GPOS: Priority Inversion

Suppose a lower priority task and let's call this task task now this task has a key to a safe, but since there are so many other higher priority tasks that are waiting to be executed the lower priority task with the key is not able to be executed and now let's say, at this point another extremely high top priority task task be Requires that key to the safe that task has and this scenario task needs the key to the safe but task has the key and task. I can't give the key because it's not able to be executed because there are other higher priority tasks that need to be executed first in this scenario, we call that as a priority because task which is a higher priority is actually waiting for task which is a lower priority but the task has a key that's what makes the priority inversion

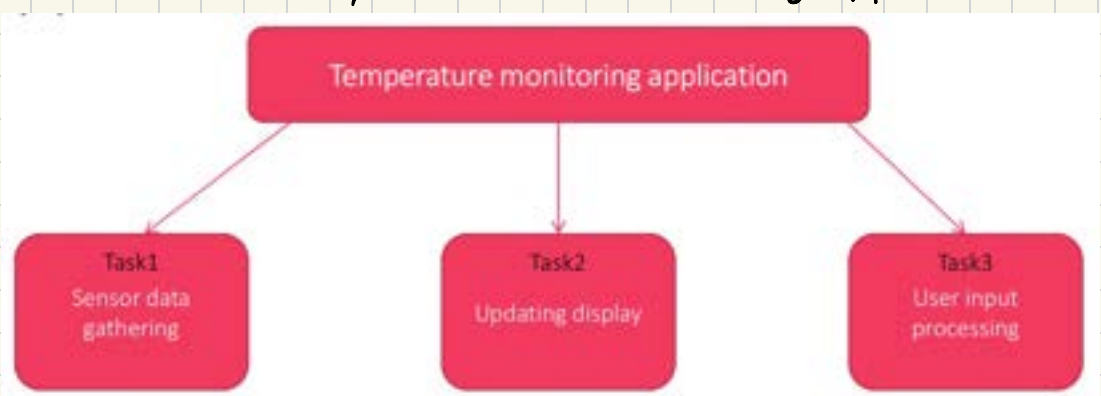
GPOS: Priority Inversion effects are insignificant
RTOS: Priority Inversions must be solved so tasks are executed within their time deadline

What are the features of an RTOS that a GPOS does not have?

- Priority based preemptive scheduling Mechanism
- None or very short Critical sections of code which disable the preemption
- Priority Inversion Avoidance
- Bounded Interrupt Latency
- Bounded Scheduling Latency, etc...

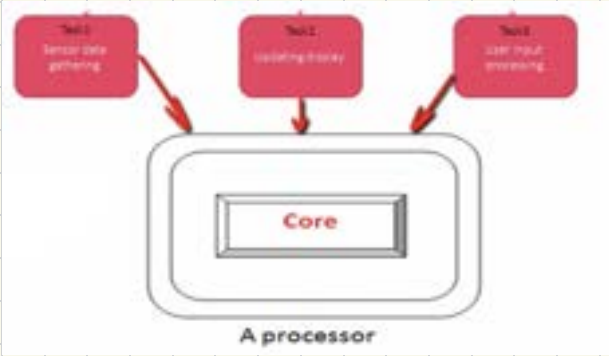
## What is Multitasking?

Application and tasks:
For example lets say we have the following Application:




A Task is just a piece of code which can be scheduled to execute by the CPU.

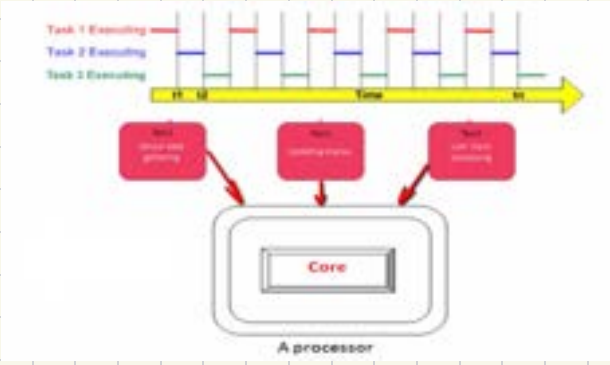
We need all 3 tasks to be running at the same time




Lets say we only have one processor with one core available. This means that the CPU can only execute one command at a time.

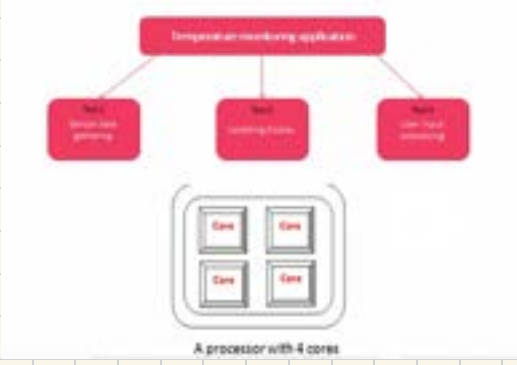
So then how do we run multiple tasks on a processor that only has one core?

We designate a small time slice for each task to run:



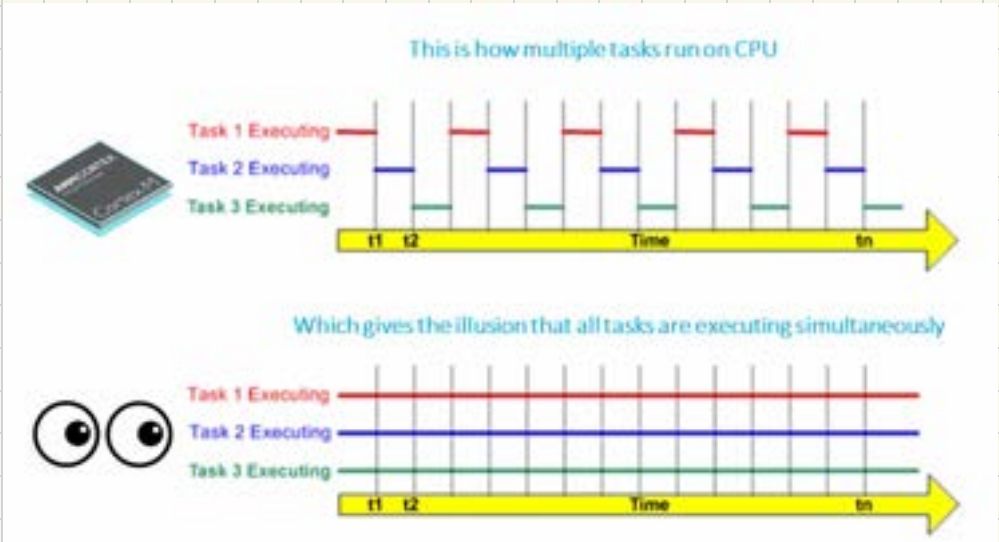

Running multiple tasks on a processor is accomplished by using a Scheduler

Another way to do this is by using a multi-core processor, each core has a responsibility to keep doing its assigned task



But this is executing one at a time, how is is simultaneous?

These intervals are very small, thus when executed at great speeds it can seem simultaneous:

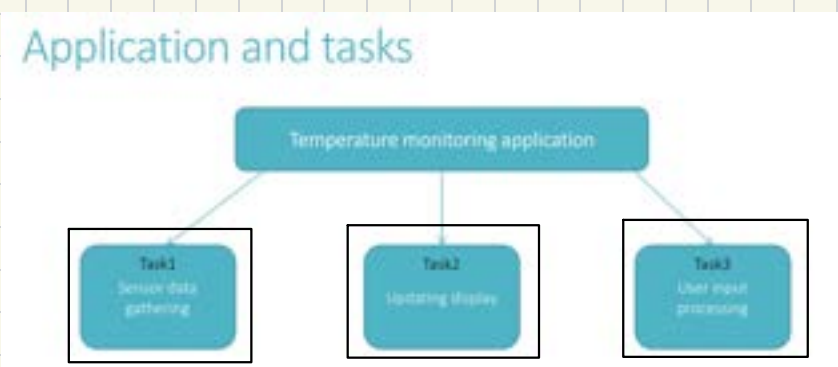



# FREERTOS TASK CREATION

## What is a Task?

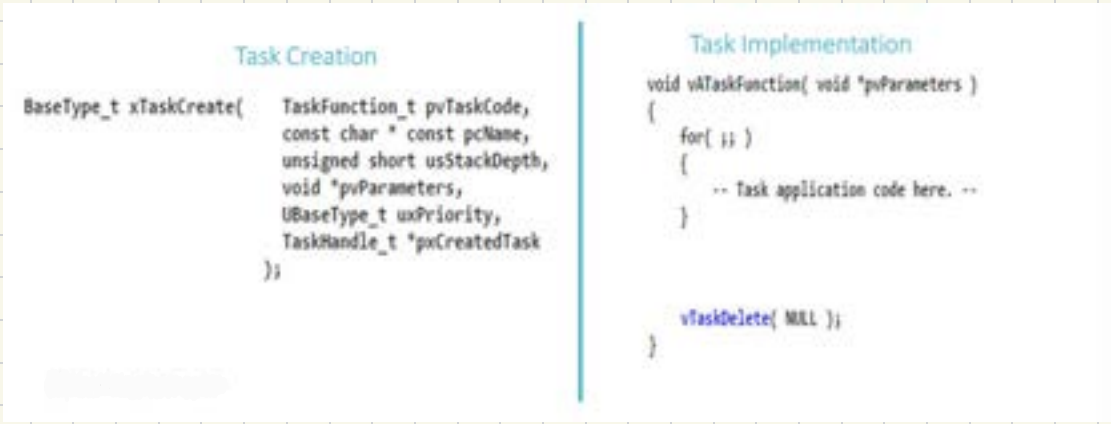
Example:

An Application can be divided into multiple tasks




A Task is nothing but a piece of code which is scheduleable

# Creating and Implementing a Task




First we create a task

↳ Task creation means creating a task in the memory

Then we create a handler which is the task implementation, also called the task function

↳ This is the function that runs on the CPU

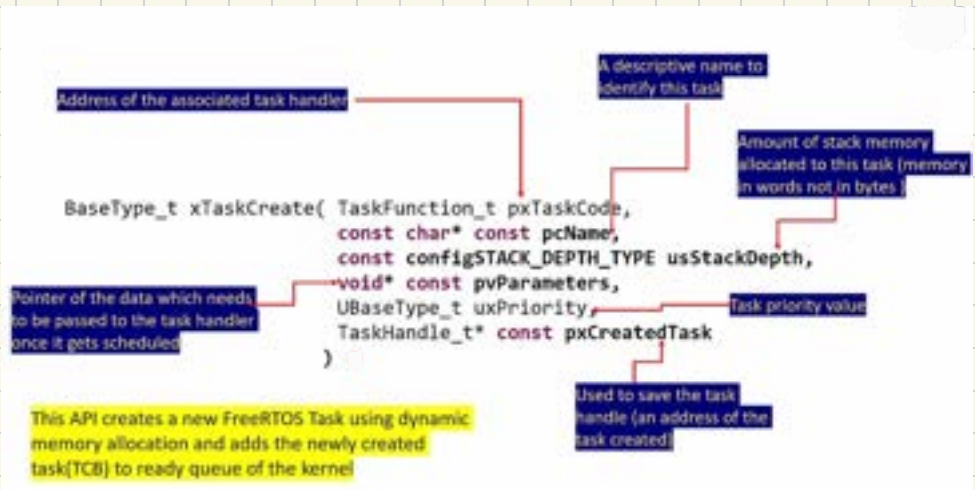
# Typical Task Handler

```
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 an 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 );
}
```

Key points

- Task Handler is generally implemented as an infinite loop
- Task handler should never return without deleting the associated stacks

FreeRTOS Task Creation API




usStackDepth

Word: A word is the maximum size of data which can be accessed by the processor in a single clock cycle.

For ex, if you are using a 32-bit MCU, then 32 bits = 4 bytes, then your word size is 4 bytes. So if you set usStackDepth = 400, then that means 400x4 = 1600 bytes = usStackDepth

## Task Priorities

- A priority value comes into the picture when there are two or more tasks in the system, in such case, the system is called a multi-task system
- The priority value helps the scheduler decide which tasks should run first on the processor
- Lower the priority value, the less importance and urgency is given to the task
- In FreeRTOS, each task can be assigned a priority value from 0 to (configMAX_PRIORITES - 1), where configMAX_PRIORITES is a macro that may be defined in the user-defined FreeRTOSConfig.h
- The user must define configMAX_PRIORITIES as per the application's needs. Using too many task priority values could lead to RAM Overconsumption. If many tasks are allowed to execute with varying task priorities, it may decrease the system's overall performance.
- A good rule of thumb is that if one is unsure of what value to set configMAX_PRIORITIES, then set it to 5. This means there are 5 levels of priority that can be assigned to any task, 0 being least important, 4 being the most important.

## Scheduling

- Lets say that you have created 2 tasks with the xTaskCreate API. After calling this function, those 2 tasks are now in the ready task list of FreeRTOS, when tasks are in this list, they are in the READY state

- If tasks are to be dispatched, so that they can run on the CPU, because as mentioned before, creating tasks only allocates memory them. In order to dispatch them, the only way for that is through the scheduler.

- The scheduler is a piece of code that is part of the FreeRTOS kernel, which runs in privileged mode of the processor

- To start the scheduler after creating tasks, call the following API function provided by FreeRTOS:

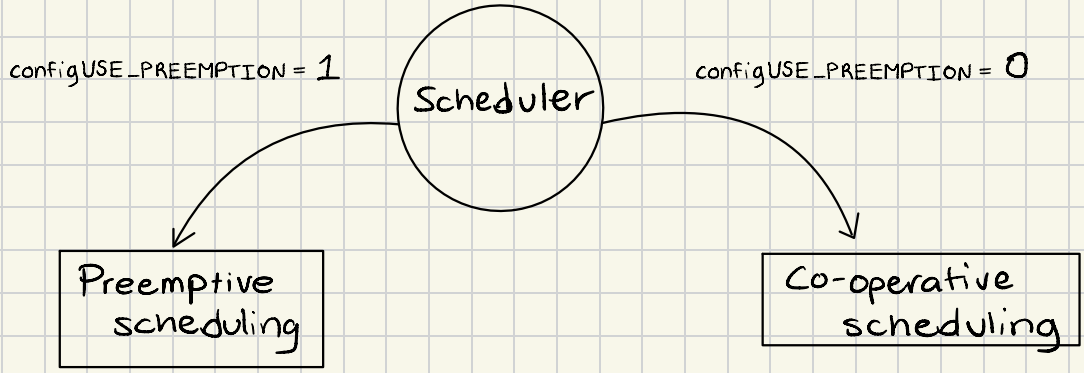
  vTaskStartScheduler()

## Scheduling Policies

The scheduler schedules tasks to run on the CPU according to the scheduling policy that the user configures. There are two types of scheduling policies:

1) Pre-emptive scheduling

2) Co-operative scheduling




configUSE_PREEMPTION is defined in FreeRTOSConfig.h, and by default it is set to 1

## Pre-emptive Scheduling

### What is pre-emption?

It means to replace a running task with another task.

During pre-emption, the running task is made to give up the processor even if it hasn't finished it's work. The scheduler does this to run some other tasks of the application.

The task that gave up the processor simply returns to ready state.

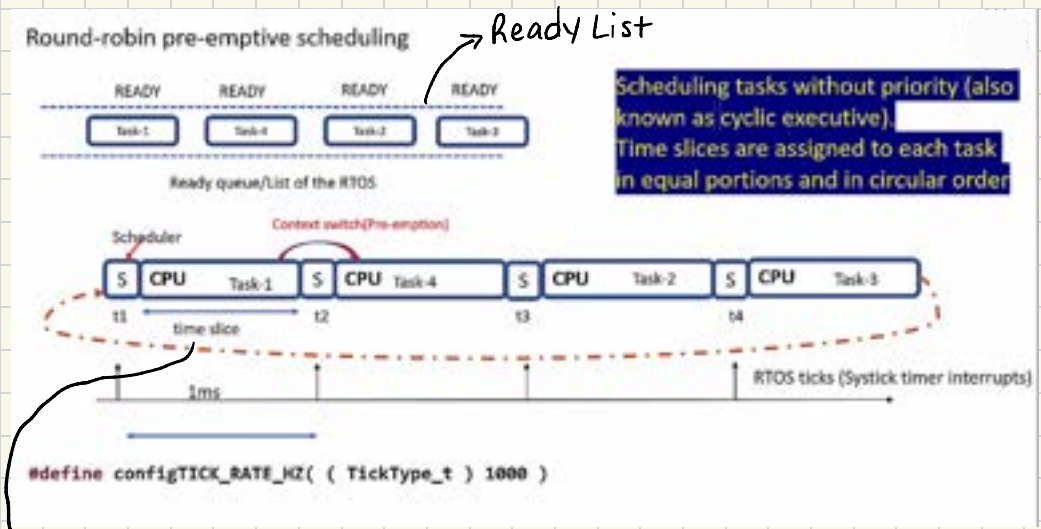
There are mainly two types of pre-emptive scheduling:

Pre-emptive Scheduling

Round Robin (Cyclic) pre-emptive scheduling

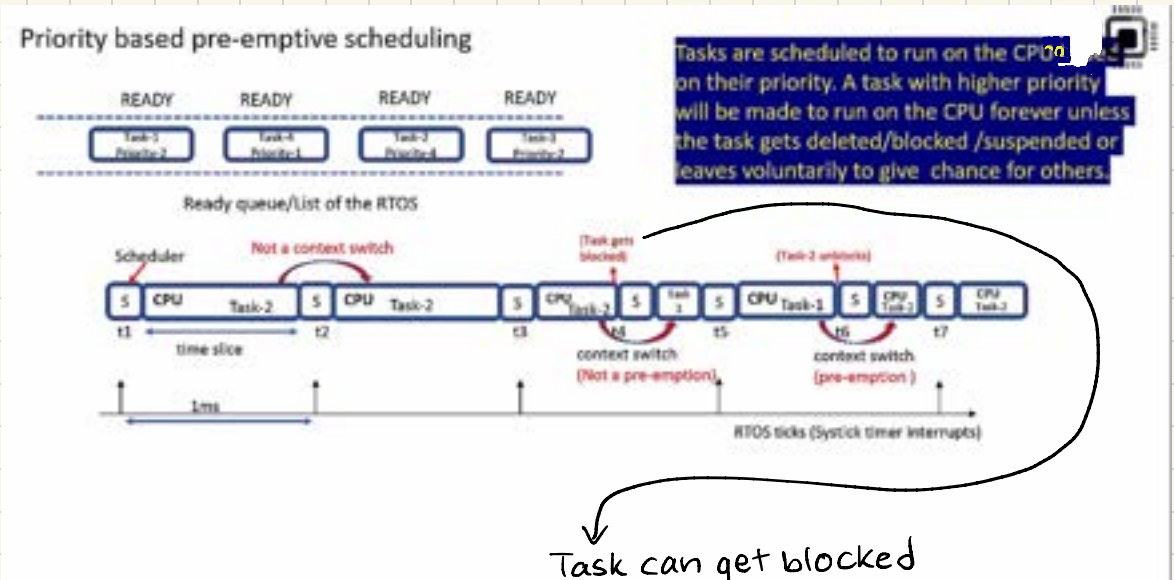
Priority-based scheduling

## Round Robin Pre-emptive Scheduling




can be any value that user defines

# Priority Based Pre-emptive Scheduling




Task can get blocked because it can be waiting for a specific event to occur and that event has not occured thus the scheduler dispatches the next highest priority task.

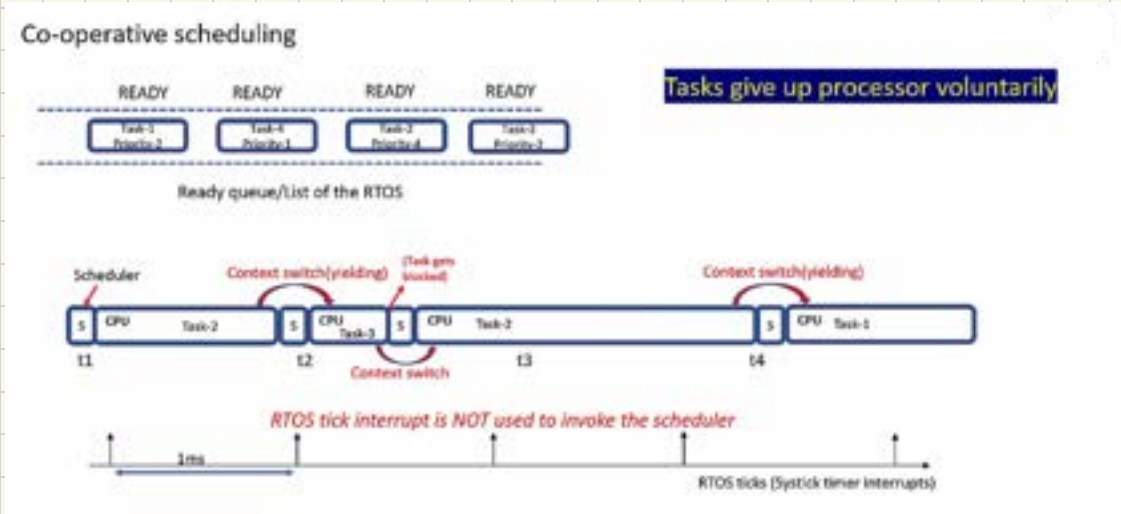
## Co-operative Scheduling

A Task cooperates with other tasks by explicitly giving up the processor, this action is called: Processor Yielding.

There is no 'pre-emption' of the tasks by the scheduler. That is, the running task will never be interrupted by the scheduler.

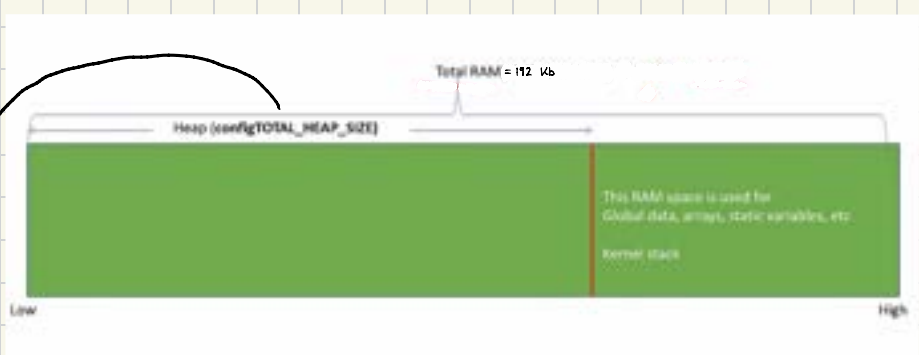
The RTOS tick interrupt does not cause any pre-emption, but the tick interrupts are still needed to keep track of the kernel's real-time tick value.

Tasks give up the CPU when they are done or periodically or when they are blocked/suspended waiting for an event or resource.



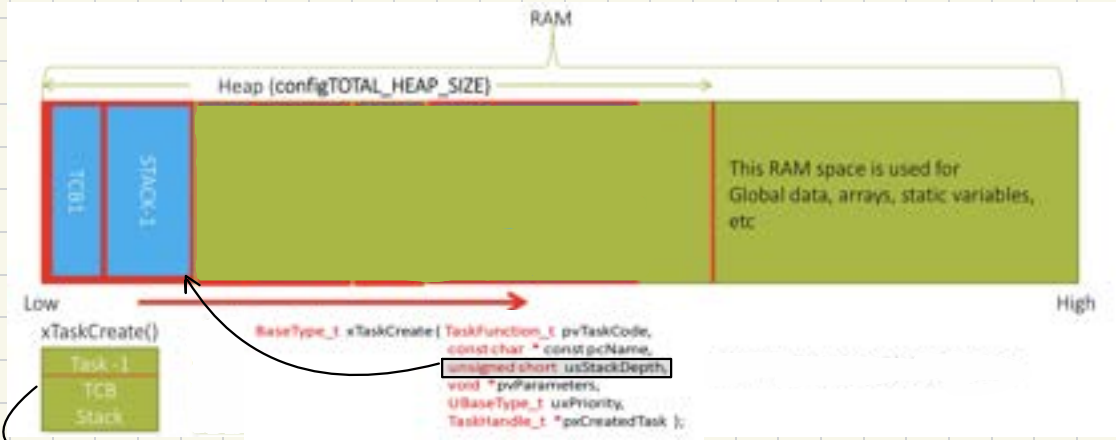

## What happens behind the scenes when you create a Task?

Lets say that our microcontroller has 192 Kb of RAM. then,



↳ This is user-defined and can be found in FreeRTOSConfig.h

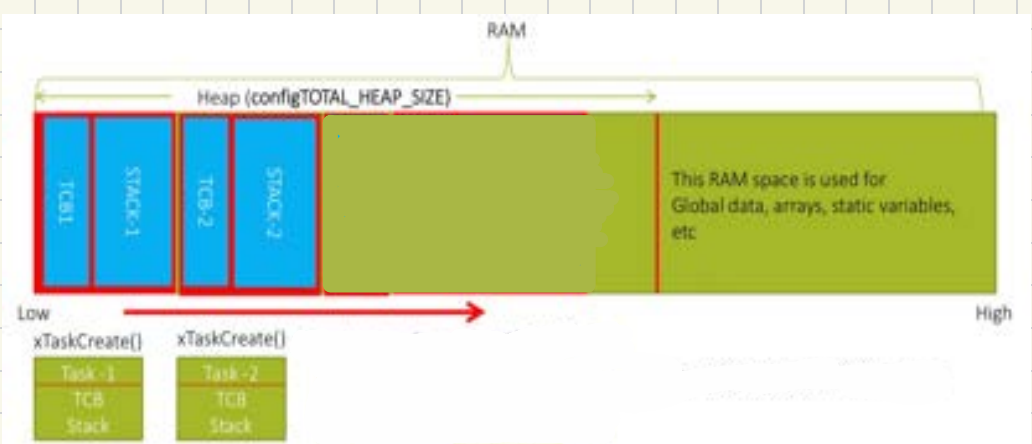
Now, lets say you create a task using xTaskCreate,



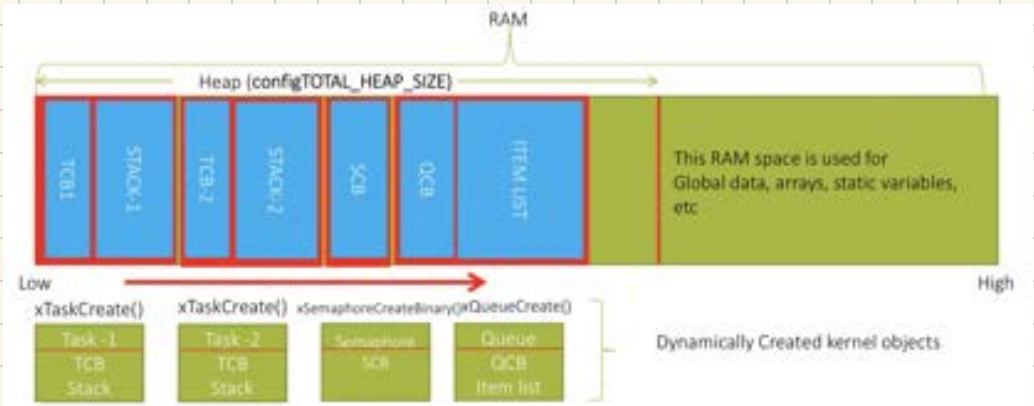

↳ The TCB is the Task Contro Block. A Task Control block is allocated for each Task, and stores information, including a pointer to the tasks contex (The task's run time environment, including register values)

The stack as we know, holds local variables and etc, that are declared in the task's implementation

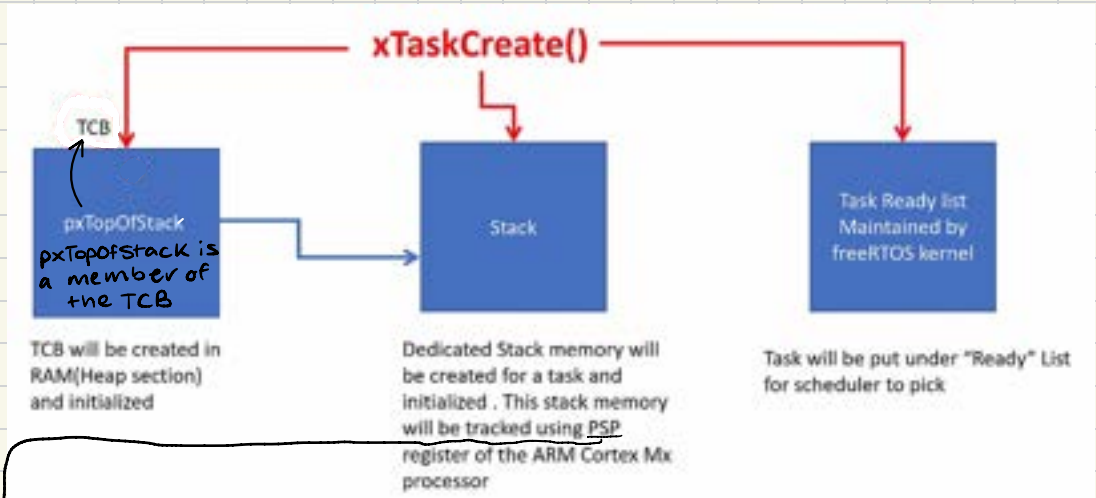
Likewise, if we create another task,



It does not only apply for tasks, when you create any FreeRTOS objects, they take up heap memory, for example:




In Summary, when you create a task, 3 important things will happen:




→ Arm Cortex Mx processor has 2 stack pointers:
PSP: Process Stack Pointer
MSP: Main Stack Pointer

# TRACE TOOL INTEGRATION

Trace tools are diagnostic tools designed to provide visibility into the operation and performance of a system. A commonly known example is SEGGER SystemView

## What is SEGGER SystemView?

The SystemView can be used to analyze how your embedded code is behaving on the target.

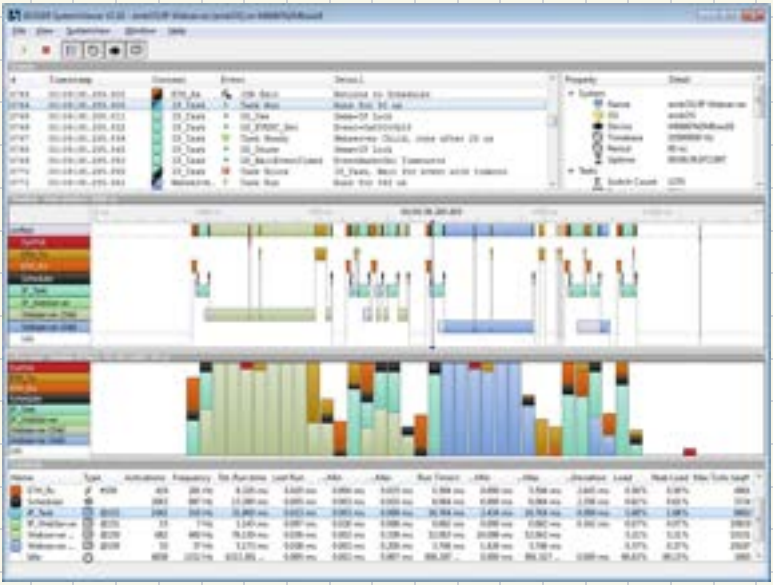
Example: In the case of an FreeRTOS Application, You can analyze the following with SystemView:

- How many tasks are running and their CPU consumption Duration
- ISR (Interrupt Service Routine) Entry and Exit timings and CPU consumption Duration
- Various Task behaviours such as: blocking, unblocking, notifying, yielding, etc.
- CPU idle time so you can consider sending CPU to sleep mode during idle time to save power.
- Total runtime behaviour of the application

The complete SEGGER SystemView toolkit has 2 parts,

1) PC Visualization Software: SystemView Host
2) SystemView target code.

The Host software provides the GUI to see all system information, ex.

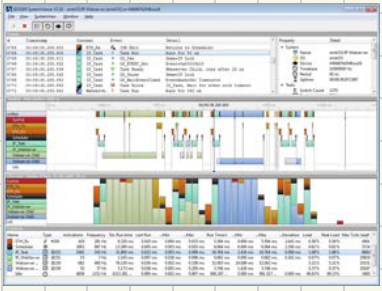


The Target code is used on the system Target (eg. microcontroller) to collect events and send them back to the PC host visualization software

SEGGER SystemView ToolKit

Host Software

SystemView Target Code

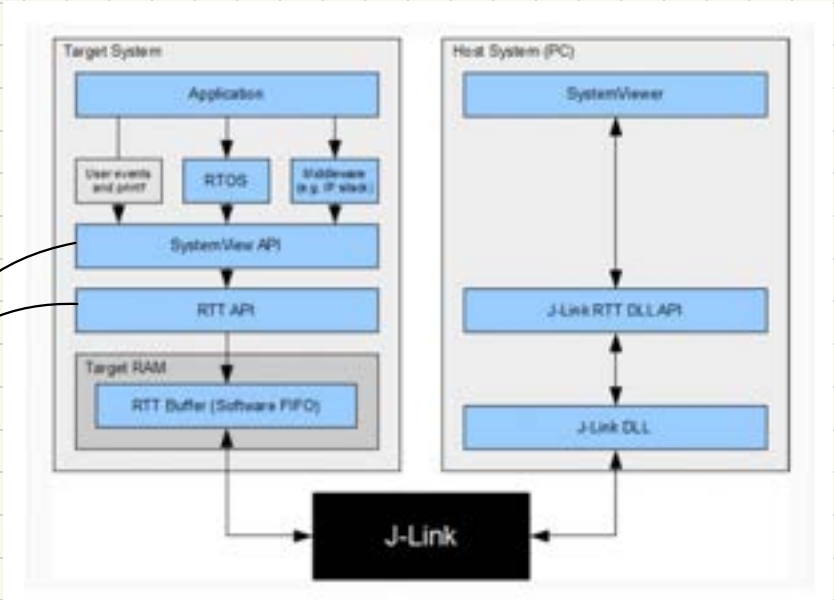


# SystemView Visualization Modes

## 1) Real time recording (continuous recording)

With a SEGGER J-Link, and it's Real Time Transfer (RTT) technology, SystemView can continuously record data, analyze and visualize it in real time.

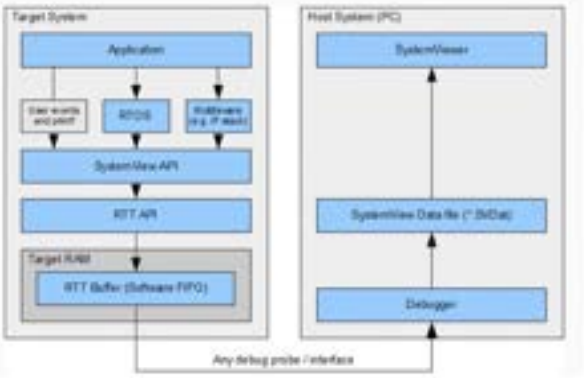
Real time mode can also be achieved via ST-Link instead of J-Link. For that, J-Link firmware must be flashed on the ST-Link circuitry of STM32 boards.




Must be explicitly Included

## 2) Single-shot recording

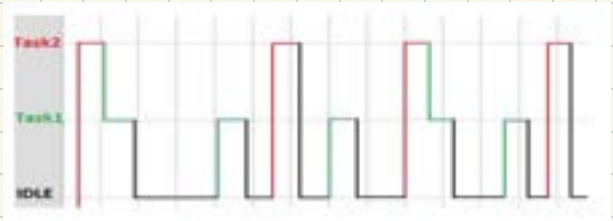
For this mode, you dont need to have J-Link OR ST-Link debugger. In this mode, the recording is started manually in the application, which allows recording only specific parts and areas of interest




when no J-Link is used, SEGGER SystemView can be used to record data until its target buffer is filled. In single-shot mode the recording is started manually in the application, which allows recording only specific parts, which are of interest.

# IDLE TASK AND SVC TASK

## Idle Task




The idle task is automatically created when the RTOS scheduler is started to ensure that there is always at least one task that is able to run.

It is created at the lowest possible priority to ensure that it does not use any CPU time if there are higher priority application tasks in the ready state.

The idle task is responsible for freeing the memory allocated by the RTOS for tasks that have already been deleted. When a task is deleted, it cannot de-allocate its memory as it has been deleted. Hence, since there must be a task to execute code, the idle task takes responsibility to free memory for those deleted tasks.

When there are no other tasks running, the idle task will always run on the CPU.

You can also give an callback function in the idle task to send the CPU to low power mode when there are no useful tasks executing.

↳ This call back function is referred to as Application hook function.

in FreeRTOS, the function is: vApplicationIdleHook()

# Timer Services Task (SVC)

This is also often called "timer daemon task"

The term "daemon" generally means a program that runs a background process

The timer daemon task deals with Software timers

This task is created automatically when the scheduler is started (its created after the idle task), and in order for the timer daemon task to be created, the option macro: "configUSE_TIMERS" must be set to 1 in the FreeRTOSConfig.h file.

The RTOS uses this daemon to manage FreeRTOS software timers and nothing else.

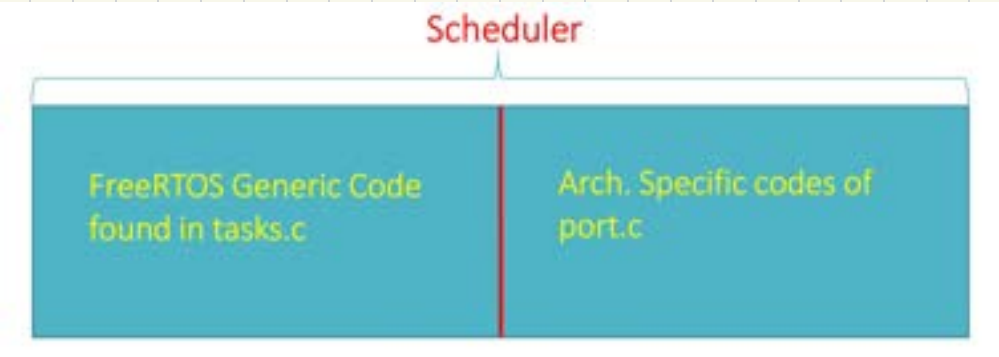
Software timers are timers defined by FreeRTOS, whereas hardware timers are timers based on actual hardware.

If you dont use software timers in your FreeRTOS application then the daemon task is not required and can be disabled. In order to disable it, you just need to "configUSE_TIMERS" = 0 in the FreeRTOSConfig.h file.

Since software timers are just normal timers, but managed by software, they can also have callback functions. These callback functions execute in the context of the timer daemon task.

# FREERTOS SCHEDULER

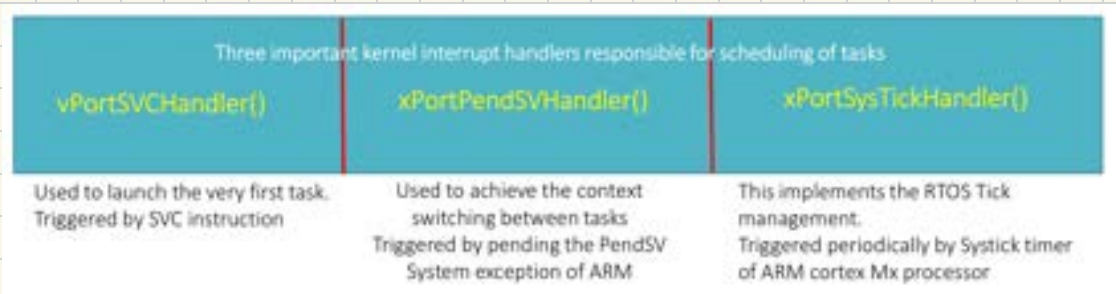
In FreeRTOS, the scheduler code is actually a combination of FreeRTOS Generic code + Architecture specific code




## Architecture specific codes responsible to achieve scheduling of tasks

All architecture specific codes and configurations are implemented in port.c and portmacro.h

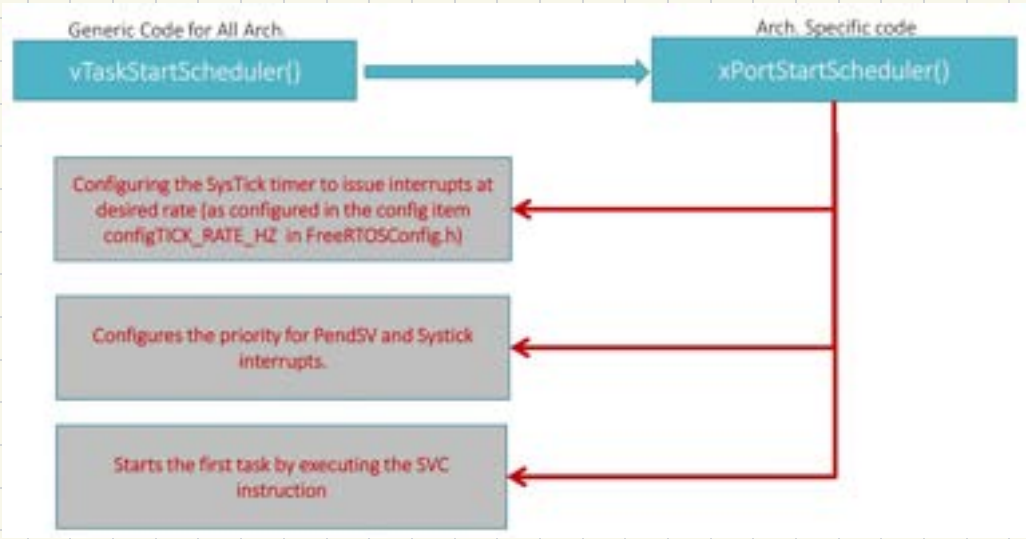
If you are using ARM Cortex Mx processor, then you should be able to locate the below interrupt handlers in port.c. These handlers are part of the scheduler implementation of FreeRTOS.




Will be discussed in detail later on

# xPortStartScheduler()

When the vTaskStartScheduler function is called, it initializes the scheduler and all architecture specific interrupts will be activated, it also creates the idle and timer daemon task and it also calls a specific function that initializes the architecture specific code. This function is called "xPortStartScheduler"



# FREERTOS AND ARM CORTEX MX ARCHITECTURE SPECIFIC DETAILS

## FreeRTOS Kernel Interrupts

When FreeRTOS runs on an ARM Cortex Mx processor-based MCU, the following hardware interrupts are used to implement the scheduling of tasks:
All of these interrupts are configured to have the lowest priority possible.

## 1. SVC Interrupt

The SVC (Supervisor Call) interrupt is a system-level interrupt used in ARM Cortex-M processors. The SVC in general is used for the following functions:

1. System Service Calls: SVC is used to transition from user mode (non-privileged) to kernel mode (privileged) safely. This is especially relevant in systems with an operating system that requires a mechanism to access system-level functions or kernel APIs safely from user applications.

2. Safe Execution Environment: SVC provides a controlled environment to execute privileged operations, which is safer than allowing direct access to system resources from user applications.

In the context of FreeRTOS, the SVC Interrupt causes the execution of the SVC instructions which in turn execute the SVC handler which then is used only once, and that is to launch the very first task.

## 2. PendSV Interrupt

The PendSV (Pendable Service Call) interrupt in ARM Cortex-M processors is specifically designed for use in operating systems, particularly Real-time operating Systems (RTOS). Its primary goal is context switching.

What is context switching?
Context switching simply means the process of switching the processor's state from one task (or thread) to another.

Here are the key functions of the PendSV Interrupt:

1. Context Switching: The primary use of PendSV in an RTOS like FreeRTOS is for context switching - changing the currently running task. When the RTOS decides that a different task should run (due to scheduling decisions like time slicing, task priority changes, or a task becoming blocked or unblocked), it sets the PendSV to trigger a context switch.

2. Deferred Execution: Because PendSV has the lowest priority, it ensures that all other higher-priority tasks and interrupts are executed first. This deferment is crucial in maintaining system responsiveness and ensuring that high-priority interrupts are not delayed by the context switching process.

3. Efficiency: PendSV can be set to trigger, but the actual context switch won't happen until all higher priority interrupts are serviced. This leads to an efficient handling of context switches, as it can be deferred until the processor is in a suitable state.

In the context of FreeRTOS, the PendSV is used to carry out context switching between tasks.

## 3. SysTick Interrupt

The SysTick (System Tick Timer) Interrupt is a built-in timer feature in ARM Cortex-M processors.

These are the primary functions of the SysTick Interrupt:

1. Timekeeping for RTOS: In an RTOS environment, the SysTick interrupt is often used for the OS tick, which is the heart of the timekeeping mechanism in the RTOS. This tick drives the scheduling of tasks, time slicing, and delays. It essentially tells the RTOS how much time has passed, enabling it to manage task switching and timing operations accurately.

2. Periodic Interrupt Generation: SysTick generates periodic interrupts at a fixed rate, which can be used for various time-related functions, such as maintaining a system tick counter, which is crucial for time-based functions like delays, timeouts, and task scheduling.

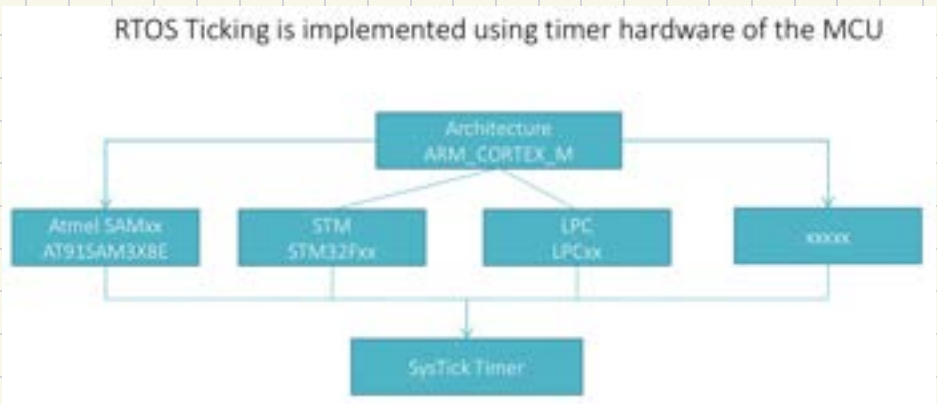
3. Simple Delays and Timeouts: Even in bare-metal (non-RTOS) systems, SysTick can be used for generating fixed delays and timeouts, providing a basic timing mechanism for the application.

In the context of FreeRTOS, the SysTick handler implements the RTOS Tick management.

If the SysTick interrupt is used for some other purpose in your application, then you may use any other timer available time peripheral.

Every interrupt (SVC, PendSV and SysTick) is configured at the lowest priority possible

# RTOS Tick (The Heart Beat)


## Why is the RTOS tick needed?

- The simple answer is to keep track of the time elapsed
- There is a global variable called "xTickCount" and it is incremented by 1 every time the tick interrupt occurs.
- RTOS ticking is implemented using SysTick timer of the ARM Cortex-Mx processor
- The tick interrupt happens at a fixed frequency defined by "configTICK_RATE_HZ" which can be found/changed in the file "FreeRTOSConfig.h"

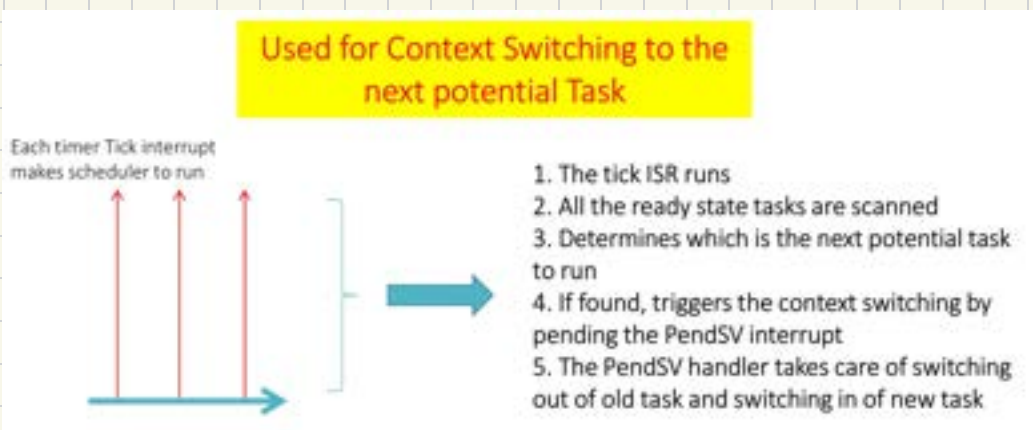
For example, lets say that we set configTICK_RATE_HZ to 1000. this means the frequency is 1000 Hz, thus the period of the tick = $1/f = 1/1000 = 1$ ms

Now lets say we have a task, and we set the task to sleep for 100 ticks, which is 100 ms because 1 tick = 100 ms


This means that the kernel has to wake up this task and schedule it on the CPU after exactly 100 ms. So how does the FreeRTOS kernel track the completion of 100 ms?

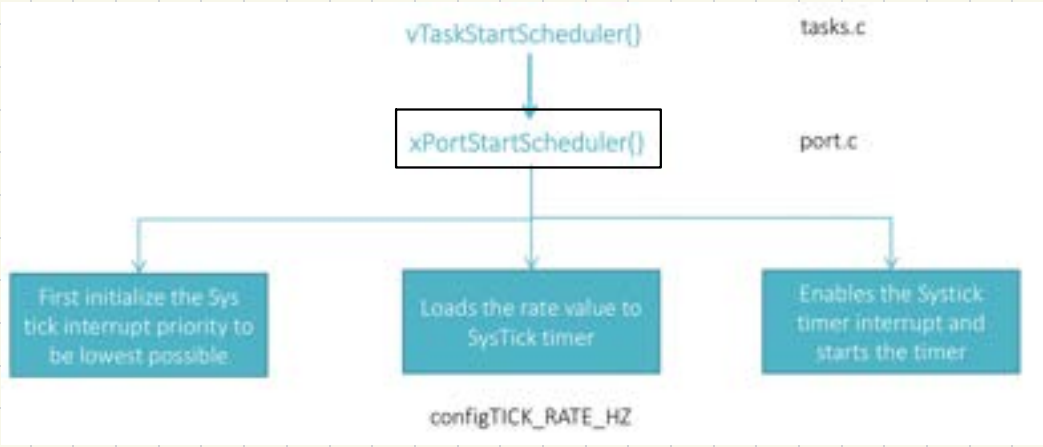
It is because of maintaining the global Tick count and incrementing it for every tick interrupt issued from the SysTick Timer.

The RTOS tick is also used for multiple other things, such as helping with context switching.



# How does RTOS Tick get configured?

We know that RTOS Tick is derived from the SysTick Timer to interrupt every 1 ms, but how does this get configured?




In the file "port.c" → vTaskStartScheduler → xPortStartScheduler → vPortSetupTimerInterrupt

↳ In this function, the counter value is loaded for the systick Interrupt

↓

Architecture Specific code (SysTick is hardware based, thus architecture based)

↳ lets say that we enter a value of 1000 for this register, this means after 1000 clock cycles an interrupt will occur, now obviously the time period of 1000 clock cycles depends on the time for one clock cycle which depends on the clock speed which is configured manually outside of FreeRTOS.

The equation in vPortSetupTimerInterrupt to load SysTick value is this:

portNVIC_SYSTICK_LOAD_REG = ( configSYSTICK_CLOCK_HZ / configTICK_RATE_HZ - 1)

configSYSTICK_CLOCK_HZ = configCPU_CLK_HZ = clock speed of MCU.

Lets say we are using a clock speed of 16MHZ, then configSYSTICK_CLOCK_HZ = 16 000 000 Hz. And lets say we want a SysTick Interrupt every 1 ms = 1000 Hz so then configTICK_RATE_HZ = 1000 Hz, then,

portNVIC_SYSTICK_LOAD_REG = ( configSYSTICK_CLOCK_HZ / configTICK_RATE_HZ - 1)

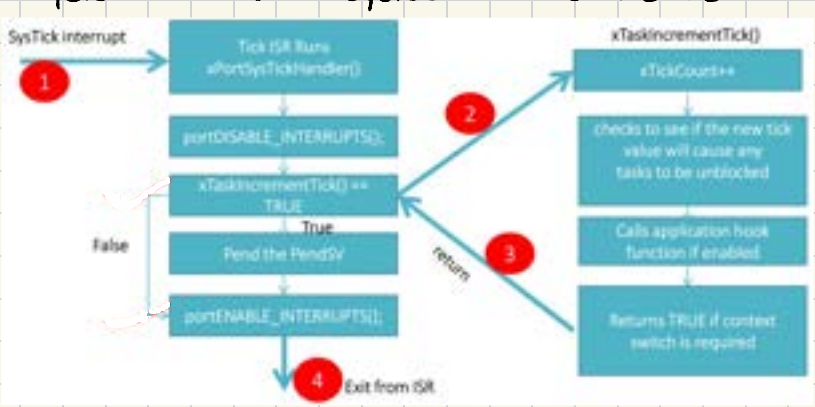
portNVIC_SYSTICK_LOAD_REG = ( 16 000 000/1000 -1)

portNVIC_SYSTICK_LOAD_REG = ( 15 999)

so now, the Systick timer will count 0 to 15 999, which equals 16 000 cycles before it generates an interrupt.

Clock speed = 16 MHz → $6.25 \times 10^{8}$ seconds or 62.5 nano seconds per cycle * 16 000 cycles = 0.0001 seconds = 1 ms

Summary:



# CONTEXT SWITCHING

## What is Context Switching?

- Context Switching is a process of switching out of one task and switching in of another task on the CPU to execute.

- In RTOS, Context Switching is taken care by the scheduler

- In FreeRTOS, Context Switching is taken care by the PendSV Handler, which can be found in port.c

## Scheduler vs PendSV Handler

In summary, while the scheduler is the decision-maker about task execution order and timing, PendSVHandler is a tool it uses to physically perform the task switching on the processor. The scheduler operates at the FreeRTOS kernel level, making high-level decisions, whereas PendSVHandler operates at a lower level, interfacing directly with the processor hardware.

- Whether a context should happen or not depends upon the scheduling policy of the scheduler.

- If the scheduler is priority based pre-emptive scheduler, then, for every RTOS tick interrupt, the scheduler will compare the priority of the currently running task with the priority of the task that is at the top of the ready task list. Whenever a task is created, it is inserted in the ready task list in a slot according to its priority, thus the ready task list is already sorted based on priority with the highest priority at the top.

Thats why during the RTOS tick interrupt, the scheduler compares the current task with the one at the top of ready list, because the task at the top of the list is the one with highest priority.

- On FreeRTOS, you can also trigger context swithces manually using the "taskYIELD" Macro

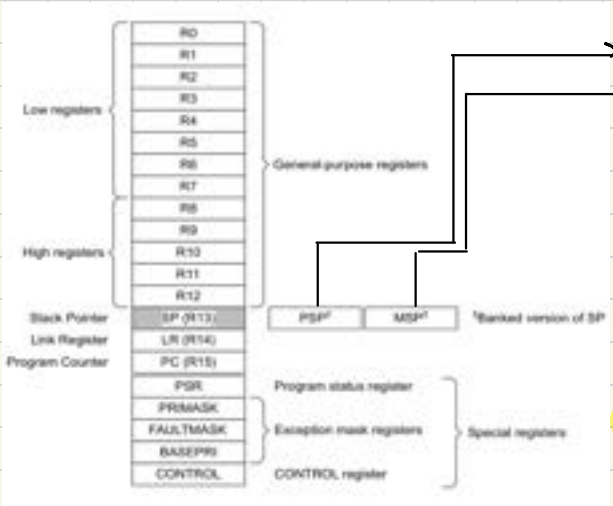
- Context switches also happen immediately whenever a new task unblocks and if the priority of the newly unblocked task is higher than the currently running task.

## Task State

When a task executes on the Processor, It utilizes the following:

- Processor Core Registers
- If a task wants to do any 'push' or 'pop' operations (during a function call) then it uses its own dedicated stack memory.

# ARM Cortex Mx Core Registers

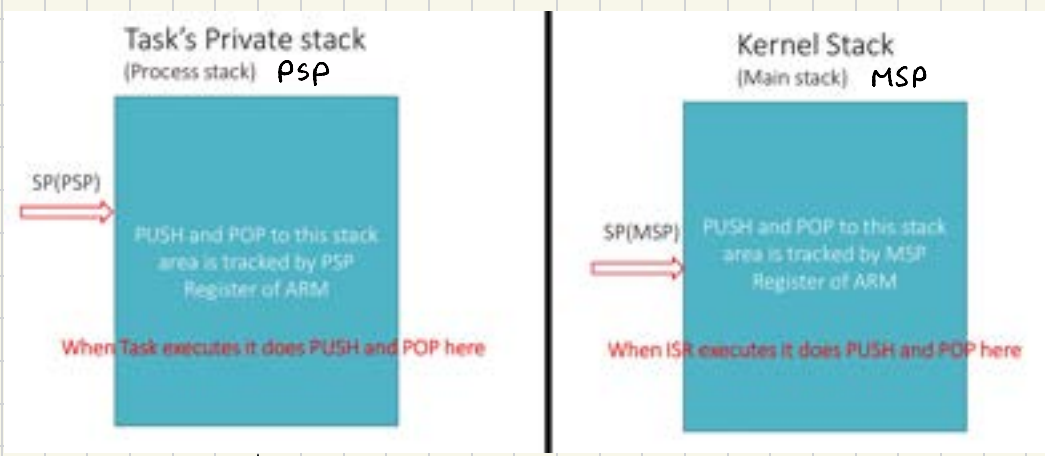



→ PSP = Process Stack Pointer

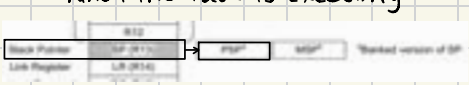
→ MSP = Main Stack Pointer

The FreeRTOS tasks actually use the PSP to track the individual task stack utilization, whereas the kernel uses the MSP in order to track the kernel stack utilization.

There are mainly 2 different stack Memories during the run time of FreeRTOS-based Application.

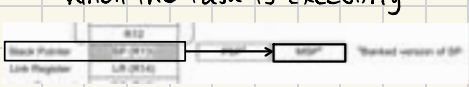



When the task is executing

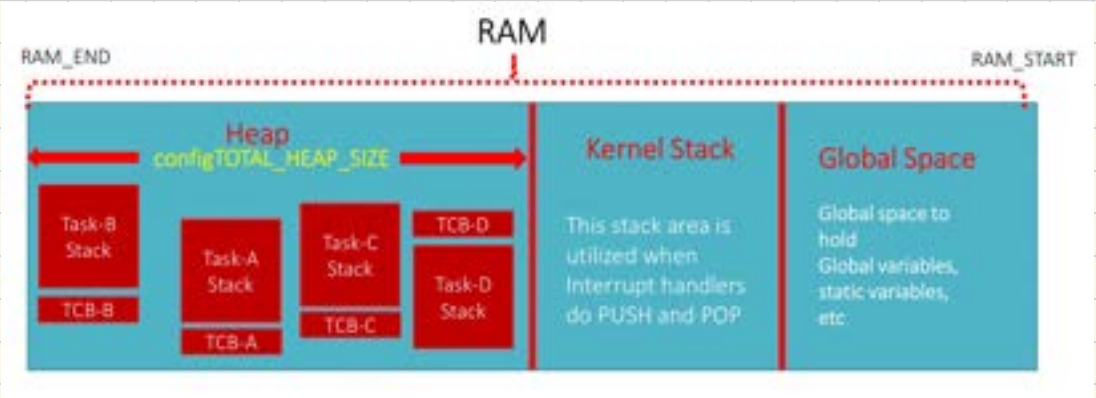


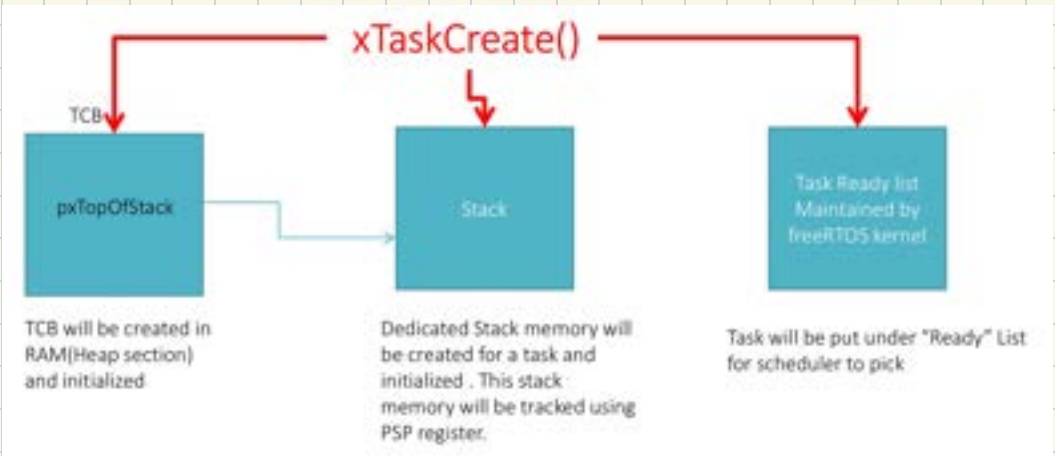
The stack pointer will be pointing to the PSP

When the task is executing



The stack pointer will be pointing to the MSP




## Task switching Out Procedure

Before a task is switched out, the following things have to be taken care of :

1) Processor core registers; R0, R1, R2, R3, R12, LR, PC, xPSR (stack frame) are saved on to the task's private stack automatically by the processor SysTick Interrupt Entry Sequence

2) If context switch is required, then the SysTick timer will pend the PendSV Exception and thus, the PendSV handler will run.

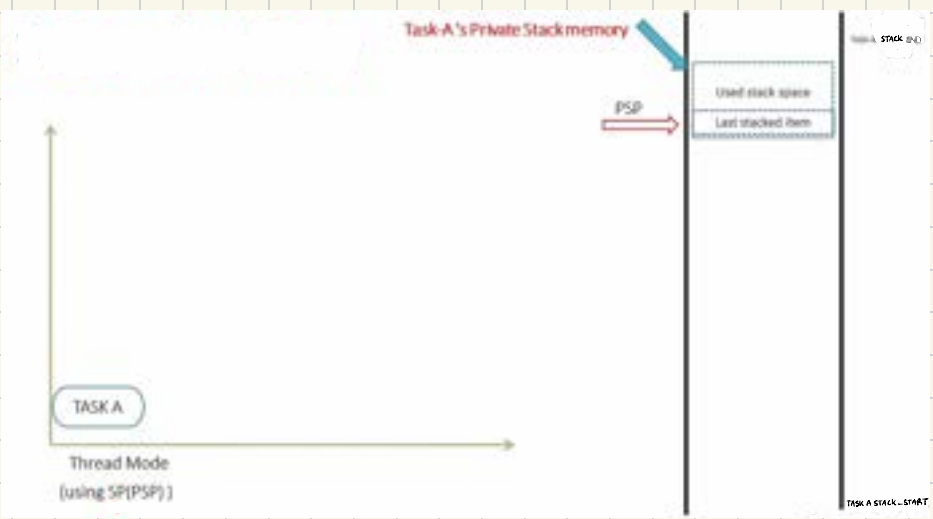
3) Processor core registers (R4-R11, R14) have to be saved manually on the tasks private stack memory (saving the context)

4) Save the new top of stack value (PSP) into the first member of the TCB.

5) Select the next potential Task to execute on the CPU. Taken care by "vTaskSwitchContext()", which is implemented in "tasks.c"

Within vTaskSwitchContext, there is a macro called "taskSELECT_HIGHEST_PRIORITY_TASK()" and this function finds the TCB of the next highest priority task.

Example:

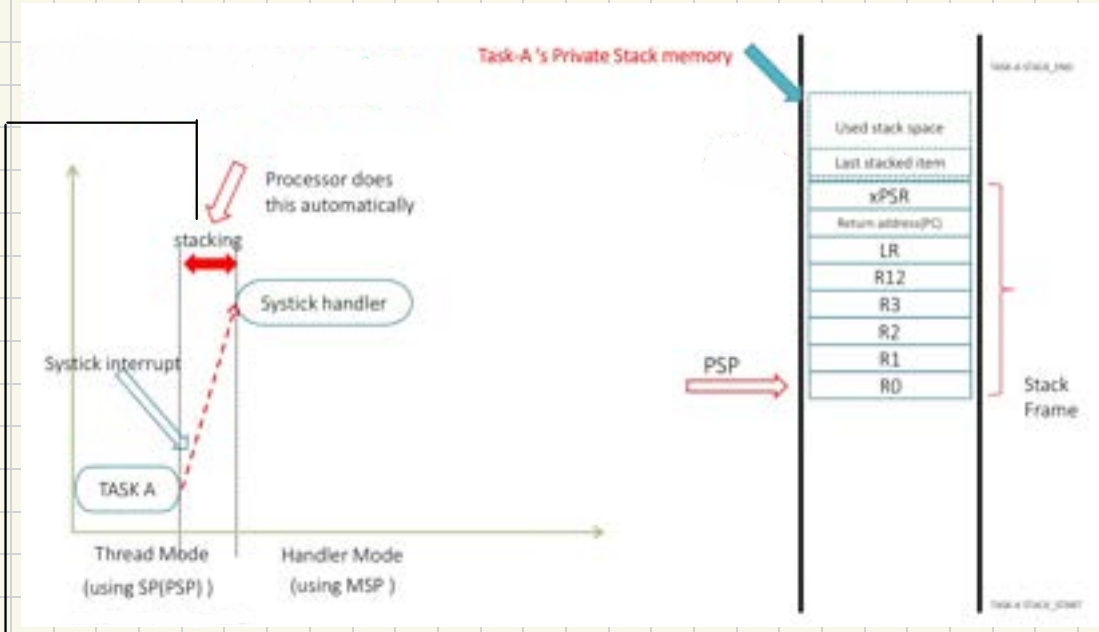



TASK A is executing normally

↓

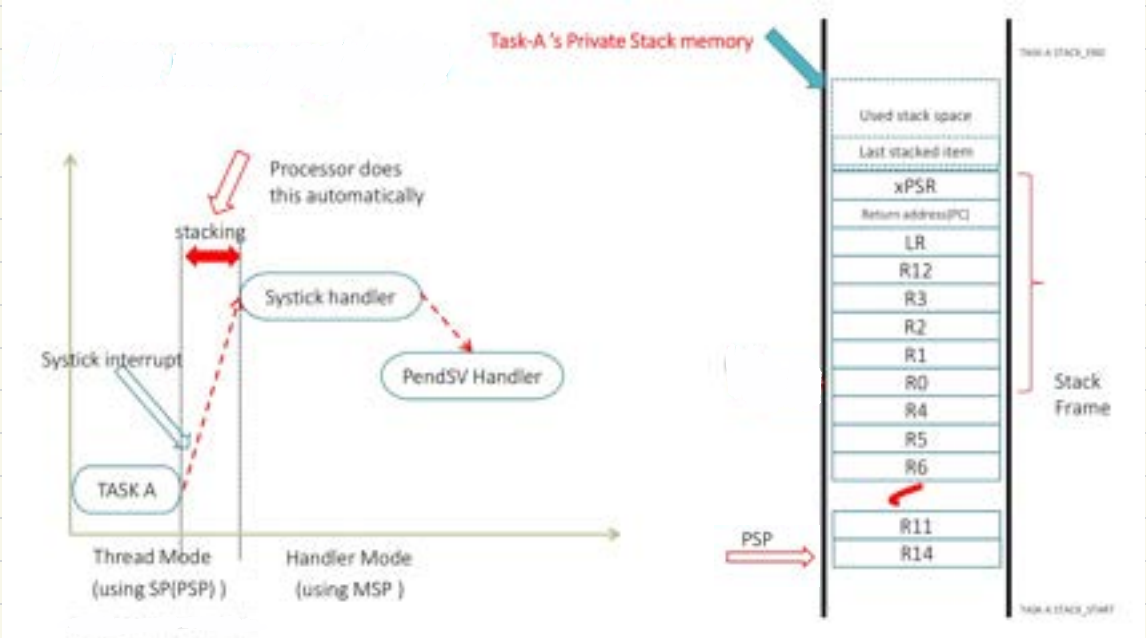
then....

A SysTick Interrupt happens:




→ During this time, the processor takes the contents from the registers: R0, R1, R2, R3, R12, LR, PC (Return Address / Program counter) and xPSR. This group of registers is referred to as the "Stack Frame" ⟶ All of these registers contain information about the current task, how far into the task has already been executed etc. The processor takes these registers and copies the data on the tasks personal stack, but there are more registers that contain valuable data which the processor does not automatically copy, such as R4-R11, R14, and the contents of these registers must be copied manually by code. (Most likely this code is already part of FreeRTOS kernel code, and this code is done in the PendSV Handler)

↓

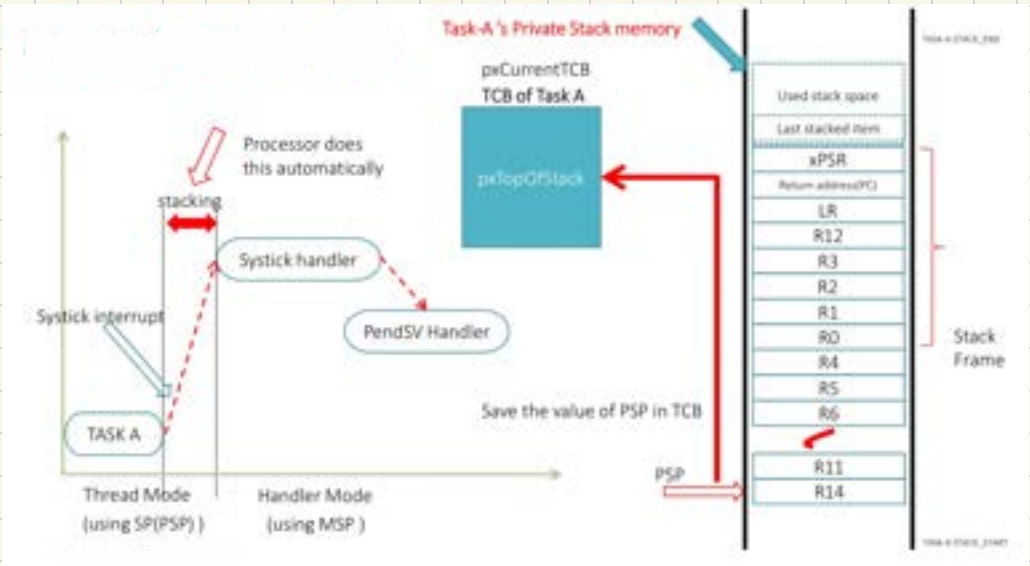



Now the PendSV Handler copied the rest of the register contents to Task A's private stack memory. At this point a new task can run and the PSP will point to the private stack of the new task,

HOWEVER!!

What happens if we did not complete the execution of the first stack and we are switching to another task? In this case, the kernel will add the current task to the ready list and its position in the ready list depends on it's priority, and then the kernel will schedule the new task and then, when the next SysTick Interrupt occurs, the scheduler will check if there any tasks that are ready to run, and if there are no tasks in the ready list with higher priority than TASK A, then the scheduler will schedule TASK A to run on the CPU, but from where will the CPU find the memory address to the private stack for TASK A? The PSP changes when we went from TASK A to TASK B?

Thats why before we switch tasks, we save the PSP for TASK A so that later on we can come back to it if we have to for whatever reason, and we save the PSP in the TCB of the Task:




## Task Switching In Procedure

After the switching out procedure is done, at the end of the switching out procedure, through the "taskSELECT_HIGHEST_PRIORITY_TASK()", the new task's TCB Address is stored and can be accessed by "pxCurrentTCB", then these final steps are done:

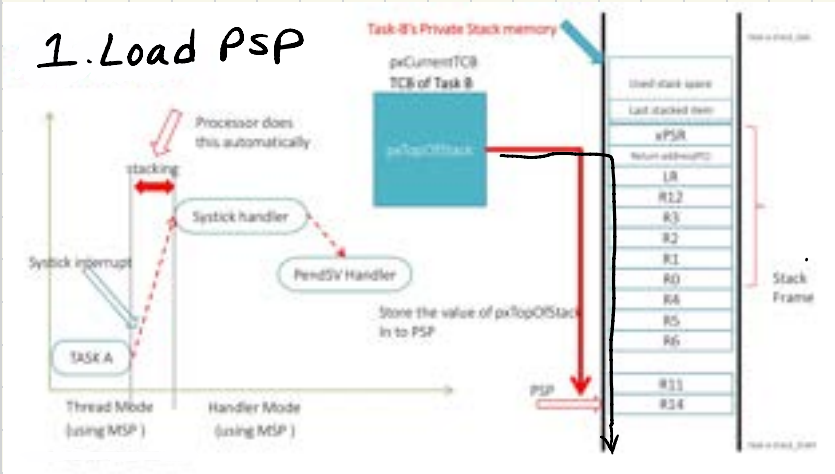
Now remember that we are switchin IN the new task, so we are taking information from the private stack of new task (popping them) and placing them onto the CPU registers.

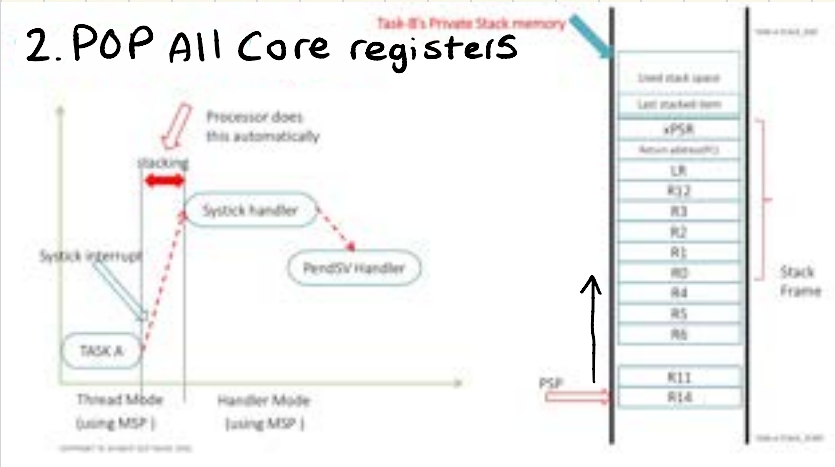
This is how its done:

1) Lets say we are switching in Task B, so firstly we will get the address of the top of TaskB's private stack and place that pxTopOfStack value in the PSP register.

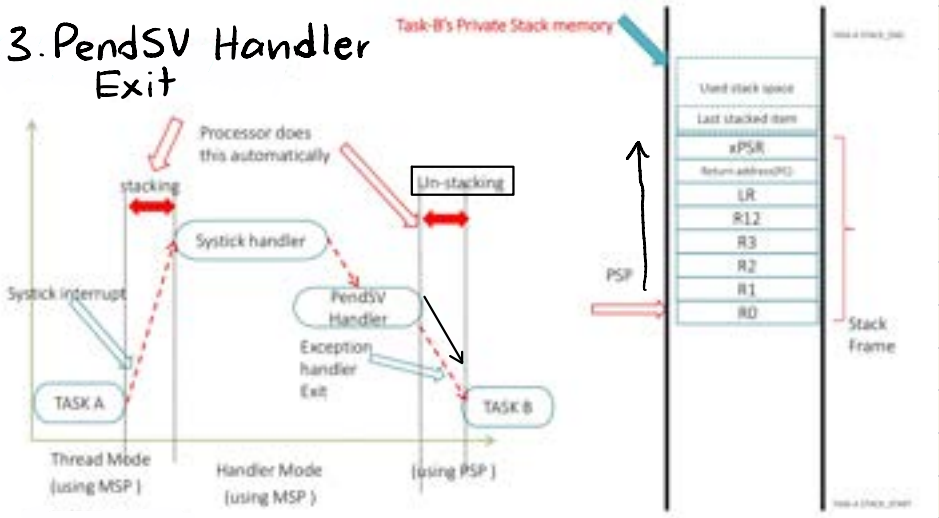
2) Then we pop (remove from stack and place into CPU register) the registers (R4-R11, R14), and this cleans the context of the stack and adds it to CPU register

3) After the pop, the PSP is pointing to the remaining registers (Stack Frame) and these registers will automatically be popped out as the PendSV Handler exits.

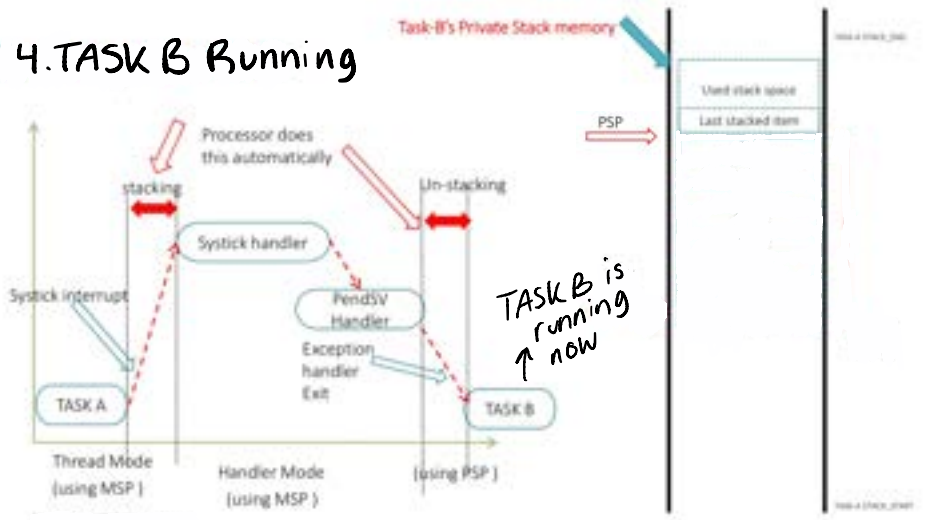


## 3. PendSV Handler Exit



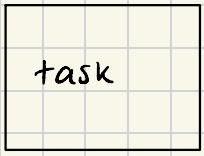

## 4. TASK B Running



# TASK STATES

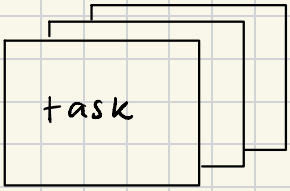
In a very simplistic Model, tasks can be one of two groups

Running



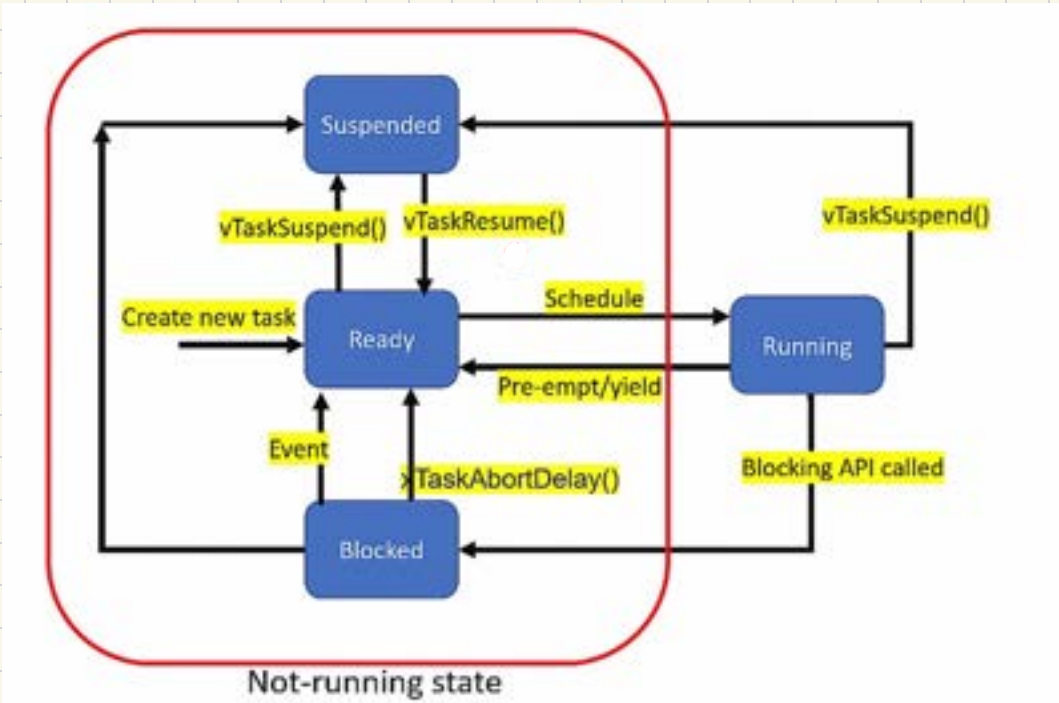

Only one task can be running at any point in time

Not Running



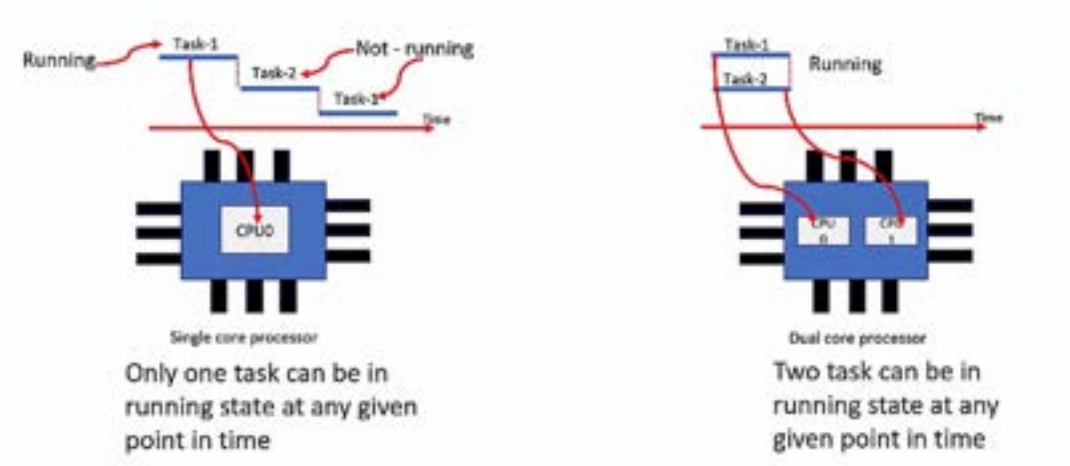

All other tasks which are not running currently on the CPU are the not-running state

↑ Was a simplified model, this is the actual model:



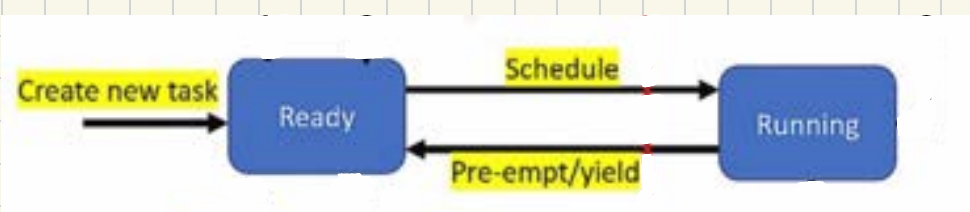
## Running State

When a task is currently executing on the processor, it is said to be in the RUNNING state.




## Ready State

- Tasks that can run on the CPU when the scheduler schedules them
- The scheduler may not schedule the ready state tasks to run on the CPU if some higher priority task is currently executing on the CPU.
- In FreeRTOS, whenever a task is created, for example by using the xTaskCreate() API, it enters the READY state



# Blocked State

- A task can leave a CPU from the RUNNING state and voluntairily choose not to run on the CPU until an internal or external event is met. This state is the BLOCKED state.

- When a task is in the BLOCKED state, it will not consume any CPU time, because it is not running, it is waiting for an internal/external event.

- In FreeRTOS, there is an API called vTaskDelay(). If a task calls vTaskDelay(500), then the task will enter the BLOCKED state for 500 RTOS ticks, after which it unblocks and enters the READY state. The event of 500 ticks being completed is an Internal temporal Event, because 500 ticks completion is caused from internal source (RTOS ticks).

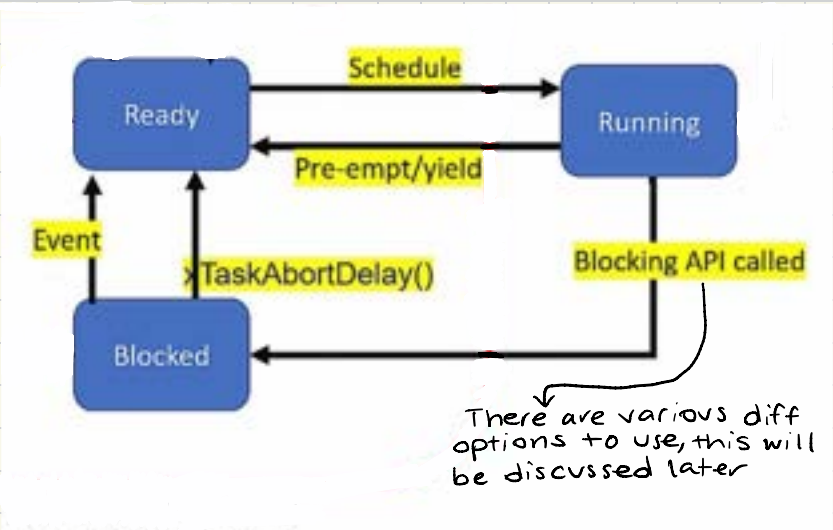
Question: What is the difference between HAL_Delay() and vTaskDelay()?

HAL_Delay(ms) halts the execution of the current task for the specified period. During this time, the CPU is essentially idle and not performing any useful work. Whereas, in vTaskDelay(ticks), it places the current task into a blocked state for the specified number of ticks. During this period, the CPU can perform tasks other than the one that has just been delayed.

- A task can also be blocked waiting on an external event, such as data arrival to the queve or it could be waiting for the availability of a lock in the case of mutex or semaphore.

° A Task in the blocked state has a timeout period, which means that it cannot in the blocked state forever. After the timeout period expires, the task unblocks and enters the ready state, even if it the event it was waiting for has not occured.

NOTE: A task can only enter the BLOCKED state from the RUNNING state.




## Advantages of blocking

1) To implement the blocking delay - For example, a task may enter the BLOCKED state in order to wait for 10 milliseconds to pass.

2) For synchronization - For example, a task may enter the BLOCKED state to wait for certain data to arrive on a queue. When another task or interrupt fills the queue, the blocked task unblocks

FreeRTOS queues, binary semaphores, counting semaphores, recursive semaphores and mutexes will be discussed in greater detail later, but for now, it is important to understand that all of these can be used to implement synchronization, and thus they support the blocking of a task.

## Blocking Task API's

vTaskDelay()
vTaskDelayUntil()
} Will be discussed later

Queues Semaphores Mutex

All of these kernel objects support API's which can block a task during operation

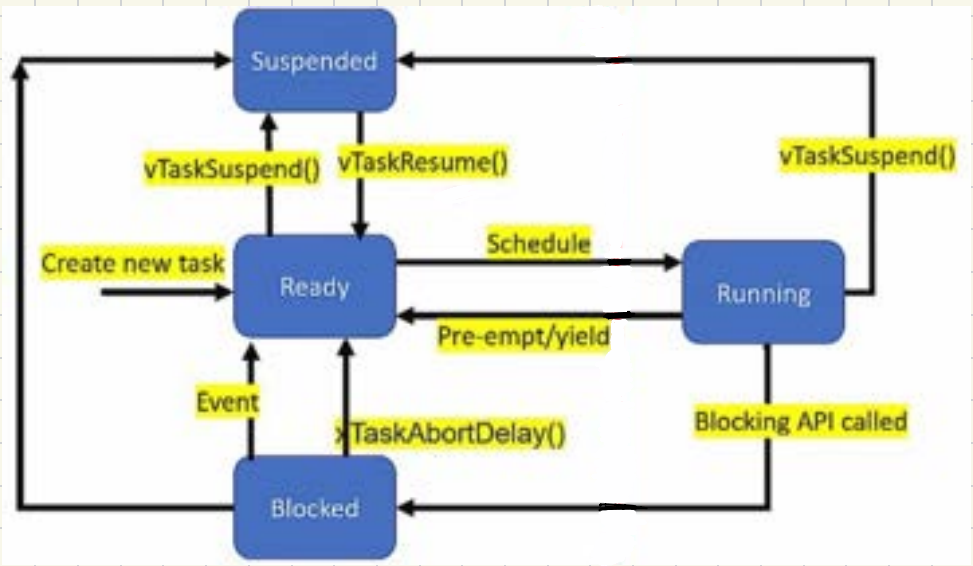
## Suspended State

- A task enters the SUSPENDED state when it or another task calls: vTaskSuspend(task_handle)
- A suspended task will not enter READY state unless some other running tasks cancel the suspension by calling the API: vTaskResume(task_handle_of_suspended_task)
- Unlike a BLOCKED task, a task in the SUSPENDED state does not have a timeout period

- A task in the READY state can enter the SUSPENDED state directly if some other running task calls the API: vTaskSuspend(task_handle), where the task_handle is the task handle of the task that is currently in the READY state and should be in the SUSPENDED state.

- A BLOCKED task can enter the READY state due to an internal or external event, however, a SUSPENDED task can ONLY enter the READY

- A READY task cannot enter BLOCKED state directly, as mentioned before. It must be scheduled to run and once it's running, then it can enter the BLOCKED state. However, a READY task can directly enter the SUSPENDED state:



# FREERTOS TASK DELAY API'S

Recall, the two API's mentioned were:
1) vTaskDelay()
2) vTaskDelayUntil()

## Why use these API's?

1) To delay a task without engaging the processor. (Replacement for 'for/while' loop-based crude delay implementation)

2) Implementation of periodic tasks

1) Lets say we created a task and inside that task we toggle an LED and then wait for 10ms, and that task repeats the toggling every 10 ms. In this case we have two main methods to implement the 10ms delay:

A) 'For' loop

↳ Lets say that if we count from 0 - 5000, this is equivalent to a 10ms delay, then in our code, after toggling the LED we can write:

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

HOWEVER!
There is a major problem with this. While the CPU is executing the 'for' loop, it cannot do anything else. The CPU is engaged

the entire loop, and we cannot execute any other tasks during this time, which sucks because the CPU is just busy waiting.

B) vTaskDelay()

Lets say we use the call: vTaskDelay(10)
This call delays the task by 10ms BUT it does NOT delay the CPU, and the CPU is not engaged during this time.

This is because vTaskDelay() is a BLOCKING delay API, and this sends the currently running task to BLOCKED state for 10ms, and as we mentioned:

> When a task is in the BLOCKED state, it will not consume any CPU time, because it is not running, it is waiting for an internal/external event.

This means that during the 10ms delay, other lower priority tasks of the system can run. After the 10ms, the task wakes up from BLOCKED state and enters the READY State.

Therefore, in RTOS Applications, in order to efficently manage multitasking, the use of 'for/while' loop-based delay is avoided since during these delays nothing useful is done but the CPU is still consumed. Using the 'for/while' loop-based delay may also prevent any lower priority tasks from taking over the CPU during the delay period.

2) Implementation of period tasks

Different types of tasks:

1) Continuous Tasks - Continuously executing
2) Periodic Tasks - Executes during a fixed period
3) Aperiodic Tasks - Executes based off events, which can occur Aperiodically
↳ Aperiodic Tasks only execute when responding to Asynchronous events

## Periodic Tasks

- Tasks which execute with a fixed execution period in the timeline

- To achieve fixed periodicity of a task, the API: vTaskDelayUntil() must be used instead of vTaskDelay()

- Fixed periodicity (task waking up and moving to READY state) is not guaranteed if vTaskDelay() is used instead of using vTaskDelayUntil()

Example how to convert ms to ticks to use the vTaskDelay and vTaskDelayUntil

### Milliseconds to ticks conversion

xTicksToWait = ( xTimeInMs * configTICK_RATE_HZ ) / 1000

Example :
If configTICK_RATE_HZ is 500, then systick interrupt(RTOS tick) is going to happen for every 2ms.
So, 500ms of delay is equivalent to 250 RTOS tick

Macro to convert ticks to ms

```
/* Converts a time in milliseconds to a time in ticks.  This macro can be
 * overridden by a macro of the same name defined in FreeRTOSConfig.h in case the
 * definition here is not suitable for your application. */
#ifndef pdMS_TO_TICKS
    #define pdMS_TO_TICKS( xTimeInMs )    ( ( TickType_t ) ( ( ( TickType_t ) ( xTimeInMs ) * ( TickType_t ) configTICK_RATE_HZ ) / ( TickType_t ) 1000U ) )
#endif
```

The next two pages contain Information about the API's, but a simple explanation is given after. The information just for reference.

# vTaskDelay

[Task Control]

task. h

```
void vTaskDelay( const TickType_t xTicksToDelay );
```

**INCLUDE_vTaskDelay must be defined as 1 for this function to be available.**

**See the RTOS Configuration documentation for more information.**

**Delay a task for a given number of ticks. The actual time that the task remains blocked depends on the tick rate.**

**The constant portTICK_PERIOD_MS can be used to calculate real time from the tick rate – with the resolution of one tick period.**

**vTaskDelay() specifies a time at which the task wishes to unblock relative to the time at which vTaskDelay() is called. For example, specifying a block period of 100 ticks will cause the task to unblock 100 ticks after vTaskDelay() is called.**

**vTaskDelay() does not therefore provide a good method of controlling the frequency of a periodic task as the path taken through the code, as well as other task and interrupt activity, will effect the frequency at which vTaskDelay() gets called and therefore the time at which the task next executes.**

**See vTaskDelayUntil() for an alternative API function designed to facilitate fixed frequency execution. It does this by specifying an absolute time (rather than a relative time) at which the calling task should unblock.**

**Parameters:**

| | |
|---|---|
| *xTicksToDelay* | The amount of time, in tick periods, that the calling task should block. |

**Example usage:**

```
 void vTaskFunction( void * pvParameters )
 {
 /* Block for 500ms. */
 const TickType_t xDelay = 500 / portTICK_PERIOD_MS;

     for( ;; )
     {
         /* Simply toggle the LED every 500ms, blocking between each toggle. */
         vToggleLED();
         vTaskDelay( xDelay );
     }
}
```

# vTaskDelayUntil

[Task Control]

task. h

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

**INCLUDE_vTaskDelayUntil must be defined as 1 for this function to be available.**

**See the RTOS Configuration documentation for more information.**

**Delay a task until a specified time. This function can be used by periodic tasks to ensure a constant execution frequency.**

**This function differs from vTaskDelay() in one important aspect: vTaskDelay() specifies a time at which the task wishes to unblock *relative* to the time at which vTaskDelay() is called, whereas vTaskDelayUntil() specifies an *absolute* time at which the task wishes to unblock.**

**vTaskDelay() will cause a task to block for the specified number of ticks from the time vTaskDelay() is called. It is therefore difficult to use vTaskDelay() by itself to generate a fixed execution frequency as the time between a task unblocking following a call to vTaskDelay() and that task next calling vTaskDelay() may not be fixed [the task may take a different path through the code between calls, or may get interrupted or preempted a different number of times each time it executes].**

**Whereas vTaskDelay() specifies a wake time relative to the time at which the function is called, vTaskDelayUntil() specifies the absolute (exact) time at which it wishes to unblock.**

**It should be noted that vTaskDelayUntil() will return immediately (without blocking) if it is used to specify a wake time that is already in the past. Therefore a task using**

**vTaskDelayUntil() to execute periodically will have to re-calculate its required wake time if the periodic execution is halted for any reason (for example, the task is temporarily placed into the Suspended state) causing the task to miss one or more periodic executions. This can be detected by checking the variable passed by reference as the pxPreviousWakeTime parameter against the current tick count. This is however not necessary under most usage scenarios.**

**The constant portTICK_PERIOD_MS can be used to calculate real time from the tick rate - with the resolution of one tick period.**

**This function must not be called while the RTOS scheduler has been suspended by a call to vTaskSuspendAll().**

**Parameters:**

| | |
|---|---|
| *pxPreviousWakeTime* | Pointer to a variable that holds the time at which the task was last unblocked. The variable must be initialised with the current time prior to its first use (see the example below). Following this the variable is automatically updated within vTaskDelayUntil(). |
| *xTimeIncrement* | The cycle time period. The task will be unblocked at time (*pxPreviousWakeTime + xTimeIncrement). Calling vTaskDelayUntil with the same xTimeIncrement parameter value will cause the task to execute with a fixed interval period. |

```
Example usage:
 // Perform an action every 10 ticks.
 void vTaskFunction( void * pvParameters )
 {
 TickType_t xLastWakeTime;
 const TickType_t xFrequency = 10;

     // Initialise the xLastWakeTime variable with the current time.
     xLastWakeTime = xTaskGetTickCount();

     for( ;; )
     {
         // Wait for the next cycle.
         vTaskDelayUntil( &xLastWakeTime, xFrequency );

         // Perform action here.
     }
 }
```

Quick Answer:

- vTaskDelay delays a task for a specified number of ticks from the moment it is called.

- vTaskDelayUntil delays a task until a specific time, based on an absolute reference, ensuring fixed periodicity.

Detailed Explanation:

Scenario for Understanding:

**Imagine a task that needs to execute every 100 milliseconds (ms).**

**1. Using vTaskDelay:**

- When you use vTaskDelay(100), the task waits for 100 ms from the moment vTaskDelay is called.
- If the task execution itself takes 10 ms, the next execution starts 110 ms after the previous execution began.
- This leads to “timing drift.” Over time, the task starts later and later than intended.

**2. Using vTaskDelayUntil:**

- With vTaskDelayUntil(&xLastWakeTime, 100), xLastWakeTime is the absolute time at which the task last started.
- Even if the task takes 10 ms to execute, vTaskDelayUntil schedules the next execution 100 ms after xLastWakeTime, maintaining exact 100 ms intervals.
- This prevents timing drift and ensures fixed periodicity.

**Why vTaskDelayUntil is Better for Fixed Periodicity:**

- Absolute Timing: It bases its delay on a fixed point in time, not on when the delay function is called. This compensates for the execution time of the task.
- Consistency: Ensures the task runs at consistent intervals, which is crucial for time-sensitive operations like sensor data sampling or communication protocols.
- Reduced Drift: It virtually eliminates timing drift, maintaining the precision of scheduling over long periods.

**Why vTaskDelay is Not as Good for Fixed Periodicity:**

- Relative Timing: It delays the task relative to when the function is called, not considering the time already spent in the task.

- Cumulative Error: Each execution can add a small delay, leading to a cumulative error over time (timing drift).

- Less Precision: For tasks requiring strict periodicity, this drift can lead to inaccuracies and unpredictable behavior.

In summary, for tasks where timing isn't critical, vTaskDelay is simple and effective. However, for tasks requiring strict timing precision and periodicity, vTaskDelayUntil is the preferred choice.

Conclusion

1. If you want to use a simple delay in your task, use the API vTaskDelay() instead of a loop-based delay, such as HAL_Delay.

2. If you want to implement a task that must execute with a precise, fixed execution period (Periodic Task) then use vTaskDelayUntil()

# FREERTOS TASK NOTIFICATIONS

## RTOS Task Notification

Each RTOS task has a 32-bit notification value which is initialized to zero when the RTOS task is created

An RTOS task notification is an event sent directly to a task. This event can unblock the recieving task and can optionally update the receiving tasks notification value in multiple different ways. For example, a notification that is sent to Task A, may overwrite Task A's notification value, or just set one or more bits in the receiving task A's notification value.

## FreeRTOS Task Notification API's

xTaskNotifyWait() – Tell a task to wait for a notification
xTaskNotify() – Send a notification event

## xTaskNotifyWait()

If a task calls xTaskNotifyWait(), then the task waits with an optional timeout until it receives a notification from some other task or interrupt handler.

↳ "Waits" means the task will be in BLOCKED state.

## xTaskNotifyWait() Prototype

```
BaseType_t xTaskNotifyWait( uint32_t ulBitsToClearOnEntry,
                            uint32_t ulBitsToClearOnExit,
                            uint32_t *pulNotificationValue,
                            TickType_t xTicksToWait );
```

| | |
|---|---|
| **ulBitsToClearOnEntry** | Any bits set in ulBitsToClearOnEntry will be cleared in the calling RTOS task's notification value on entry to the xTaskNotifyWait() function (before the task waits for a new notification) provided a notification is not already pending when xTaskNotifyWait() is called.<br><br>For example, if ulBitsToClearOnEntry is 0x01, then bit 0 of the task's notification value will be cleared on entry to the function.<br><br>Setting ulBitsToClearOnEntry to 0xffffffff (ULONG_MAX) will clear all the bits in the task's notification value, effectively clearing the value to 0. |
| **ulBitsToClearOnExit** | Any bits set in ulBitsToClearOnExit will be cleared in the calling RTOS task's notification value before xTaskNotifyWait() function exits if a notification was received.<br><br>The bits are cleared after the RTOS task's notification value has been saved in *pulNotificationValue (see the description of pulNotificationValue below).<br><br>For example, if ulBitsToClearOnExit is 0x03, then bit 0 and bit 1 of the task's notification value will be cleared before the function exits.<br><br>Setting ulBitsToClearOnExit to 0xffffffff (ULONG_MAX) will clear all the bits in the task's notification value, effectively clearing the value to 0. |
| **pulNotificationValue** | Used to pass out the RTOS task's notification value. The value copied to *pulNotificationValue is the RTOS task's notification value as it was before any bits were cleared due to the ulBitsToClearOnExit setting.<br><br>If the notification value is not required then set pulNotificationValue to NULL. |
| **xTicksToWait** | The maximum time to wait in the Blocked state for a notification to be received if a notification is not already pending when xTaskNotifyWait() is called.<br><br>The RTOS task does not consume any CPU time when it is in the Blocked state.<br><br>The time is specified in RTOS tick periods. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds into a time specified in ticks. |

| return value | pdTRUE if a notification was received, or a notification was already pending when xTaskNotifyWait() was called.<br><br>pdFALSE if the call to xTaskNotifyWait() timed out before a notification was received. |
|---|---|

## xTaskNotify()

xTaskNotify() is used to send an event directly to and potentially unblock an RTOS, and optionally update the receiving task's notification value in one of the following ways:

- Write a 32-bit number to the notification value
- Add one (increment) to the notification value
- Set one or more bits in the notification value
- Leave the value unchanged

This function must not be called from an interrupt service routine (ISR). For this case, xTaskNotifyFromISR() should be used.

## xTaskNotify() Prototype

```
BaseType_t xTaskNotify( TaskHandle_t xTaskToNotify,
                        uint32_t ulValue,
                        eNotifyAction eAction );
```

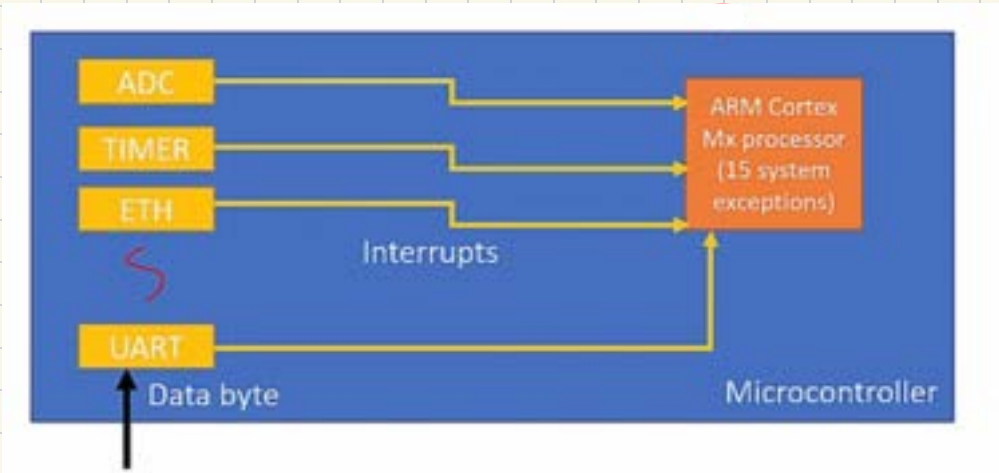
| | |
|---|---|
| **xTaskToNotify** | The handle of the RTOS task being notified. This is the *target* task.<br><br>To obtain a task's handle create the task using xTaskCreate() and make use of the pxCreatedTask parameter, or create the task using xTaskCreateStatic() and store the returned value, or use the task's name in a call to xTaskGetHandle().<br><br>The handle of the currently executing RTOS task is returned by the xTaskGetCurrentTaskHandle() API function. |
| **ulValue** | Used to update the notification value of the target task. See the description of the eAction parameter below. |
| **eAction** | An enumerated type that can take one of the values documented in the table below in order to perform the associated action. |

| **eAction Setting** | **Action Performed** |
|---|---|
| eNoAction | The target task receives the event, but its notification value is not updated. In this case ulValue is not used. |
| eSetBits | The notification value of the target task will be bitwise ORed with ulValue. For example, if ulValue is set to 0x01, then bit 0 will get set within the target task's notification value. Likewise if ulValue is 0x04 then bit 2 will get set in the target task's notification value. In this way the RTOS task notification mechanism can be used as a light weight alternative to an event group. |
| eIncrement | The notification value of the target task will be incremented by one, making the call to xTaskNotify() equivalent to a call to xTaskNotifyGive(). In this case ulValue is not used. |
| eSetValueWithOverwrite | The notification value of the target task is unconditionally set to ulValue. In this way the RTOS task notification mechanism is being used as a light weight alternative to xQueueOverwrite(). |
| eSetValueWithoutOverwrite | If the target task does not already have a notification pending then its notification value will be set to ulValue.<br>If the target task already has a notification pending then its notification value is not updated as to do so would overwrite the previous value before it was used. In this case the call to xTaskNotify() fails and pdFALSE is returned.<br><br>In this way the RTOS task notification mechanism is being used as a light weight alternative to xQueueSend() on a queue of length 1. |

# ARM CORTEX-M INTERRUPT PRIORITY AND FREERTOS TASK PRIORITY

## Hardware Priority vs Task Priority

Hardware priority is the priority values assigned to various interrupts and system executions of the processor



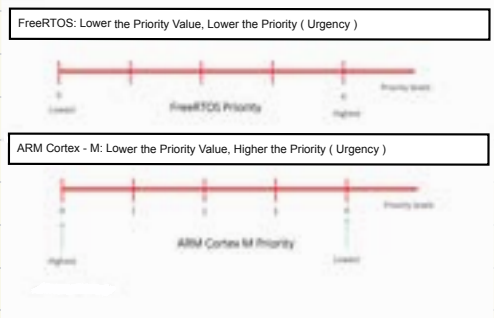

The Interrupt priorities are specific to the hardware whereas the FreeRTOS Task priorities are specific to the FreeRTOS kernel. These tasks run in thread mode of the processor, thus the tasks have a process context. Interrupt Service Routines (ISR's) / Interrupt handlers run in handler mode of the processor, thus they have an Interrupt context.

For example, let's say we are using two interrupts in our application:

Interrupt - 1 (ADC Periph) → Priority = 2

Interrupt - 2 (Timer Periph) → Priority = 5

When using ARM Cortex-M Processor, then, for hardware interrupts, Lower the priority value, higher the importance / urgency. Thus, in our application, the ADC has higher priority than the Timer.




FreeRTOS tasks run in thread mode of the processor and hardware interrupts run in handler mode of the processor. Handler mode always takes priority over thread mode, so if a task is running, and a hardware interrupt occurs, even if the priority of the hardware interrupt is lower than the task, the CPU will stop the task, change from thread mode to handler mode, execute the Interrupt service routine, and then change back to thread mode and continue executing the task.

## FreeRTOS Hardware Interrupt Configurable Items

1. `configKERNEL_INTERRUPT_PRIORITY`
2. `configMAX_SYSCALL_INTERRUPT_PRIORITY`

1. `configKERNEL_INTERRUPT_PRIORITY`

This config item decides the priority for the kernel Interrupts.

## What are the kernel Interrupts?

1) SysTick Interrupt – helps manage tick count
2) PendSV Interrupt – helps initiate and execute context switches
3) SVC Interrupt – helps launch the very first task

So what is the lowest and highest interrupt priority values possible in a ARM Cortex-M based microcontroller?

The answer can be found in the macro: `__NVIC_PRIO_BITS`

This macro can be found in the CMSIS Folder, for example, if we are using STM32F407xx MCU which is ARM Cortex-M based, then the macro can be found:

`Drivers/CMSIS/Device/ST/STM32F4XX/Include/stm32f407xx.h`

If that macro is set to 4 bits, then we have 16 total priorities that we can choose from, $2^4 = 16$, 0-15, if that macro is set to x bits, then we have $2^x$ possible priorities.

REMINDER:
In ARM Cortex-Mx processors, higher the priority value, the lower the priority (Urgency).

For ex. A Timer interrupt that has got a priority value of 15 is less prioritized (or less urgent) interrupt compared to a UART interrupt whose priority value is 2

**2. configMAX_SYSCALL_INTERRUPT_PRIORITY**

FreeRTOS API's that end with "FromISR" are special API functions are meant to only be called during hardware Interrupts.
This is a threshold priority limit for those interrupts which use FreeRTOS API's which end with "FromISR"

Interrupts which use FreeRTOS API's ending with "FromISR", should not use a priority greater than the value for this Syscall Interrupt priority macro.

Recall: Greater Priority = lesser priority value

## Example

Lets say we are using ARM Cortex Mx, and we defined our macros, Also `__NVIC_PRIO_BITS` is 3 bits in this case, so total $2^3 = 8$ different possible priorities.

```
configKERNEL_INTERRUPT_PRIORITY          = 111 ( Binary ) = 0x7 ( Hex )

configMAX_SYSCALL_INTERRUPT_PRIORITY     = 010 ( Binary ) = 0x3 ( Hex )
```

We have two categories, value and priority, Recall: Greater value = Lesser priority.

Kernel Priority = 0 since value = 7
Syscall Priority = 4 since value = 3

| Value | Priority |
|---|---|
| 0 | 7 |
| 1 | 6 |
| 2 | 5 |
| 3 | 4 |
| 4 | 3 |
| 5 | 2 |
| 6 | 1 |
| 7 | 0 |

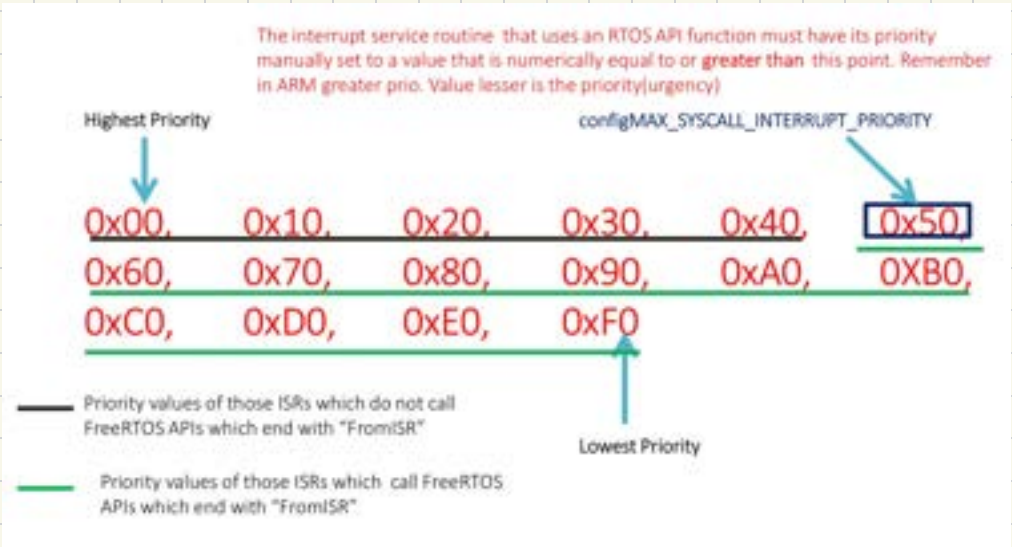
→ Interrupts running at these priority levels will never be delayed from executing because of anything that the FreeRTOS kernel is doing. Thus these levels are suitable for critical-hardware interrupts.

→ ISR's that call API functions ending with "FromISR" can use these interrupt priorities

→ Kernel interrupts have lowest interrupt priority to ensure they dont get in the way for hardware interrupts

Interrupts that do not call any FreeRTOS.org API functions can use all these interrupt priorities

Now, for a 4-bit priority possibilities = $2^4$ = 16 possible priorities




## Concluding points

- FreeRTOS API's that end with "FromISR" are special API's that are interrupt safe, but even these API's should not be called from ISR's that have priorities (urgency) above the priority defined by:

```
configMAX_SYSCALL_INTERRUPT_PRIORITY
```

- Therefore, any interrupt service routine that uses an RTOS API function must have it's priority valve manually set to a valve that is numerically equal or greater than:

```
configMAX_SYSCALL_INTERRUPT_PRIORITY
```

° Cortex-M interrupts, when created, default to having a priority value of zero. Recall: Zero is the highest possible priority valve. Therefore, never leave the priority of an interrupt that uses the RTOS interrupt-safe API's at its default value.

## FreeRTOS "FromISR" API's

FreeRTOS provides separate API's to use from interrupt handlers. These API's all end with "FromISR", for example:

`xTaskNotifyFromISR`

If a FreeRTOS API does not end with "FromISR", then it should not be used in an interrupt context (Inside ISR) and should be used in its appropriate location, maybe like inside a task or something

This does not only apply to notifications, for example, if a task wants to send something to a Queue, then it can use `xQueueSend( )`, but this API should not be used in an Interrupt, instead, `xQueueSendFromISR( )` should be used.

# FREERTOS INTERRUPT SAFE API'S

These API's are also called "FromISR" API's as all of these API's names end with "FromISR" to signify that they are safe to call from a ISR (Interrupt Service Routine).

Thus, the API's are meant to be used from an Interrupt handler. The terms Interrupt handler and ISR mean the same and are used interchangeably.

## Why have separate "FromISR" API's from normal API's?

1) It makes APIs implementation easier

When called from a task context, many RTOS APIs can put the calling task to a blocked state, but the same cannot happen when called from interrupt handlers because you cannot put the interrupt handler or interrupt context code to a blocked state. The blocked state is for tasks, not for interrupt handlers.

So now, if there were no distinct "FromISR" API's, every API must determine which context it was being called, interrupt or task context, and then carry out the function based on the context, even if each context requires a completely different procedure and environment.

The approach of combining "FromISR" functionality with normal API's exponentially increases overhead, and some architectures do not support, or allow the possibility of determing the context (Task or Interrupt context).

To overcome this problem, FreeRTOS provides separate APIs that are to be called from interrupt handlers and prohibits using APIs which dont end with "FromISR" inside an interrupt handler.

2) API function parameters would become confusing.

Suppose that we would have one API to accomplish something for both interrupt and task contexts. Now, since interrupts and tasks are fundamentally very different, each context has its own configurable options which are used as parameters to this one function.

It would be very confusing to have multiple parameters designed to be used in specific contexts for one function.

Disadvantages of using separate API's

1) The application writer must redesign the logic when there is a need to call a third-party function from an interrupt handler that uses FreeRTOS APIs which dont end with "FromISR"

For example if I have a Interrupt handler, and in this handler I call function x. If function x uses any non-"FromISR" functions, like xQueueSend(), then I cannot use function x in my interrupt handler.

A possible solution to this problem could be to create a task that does what function x does and in the interrupt handler use the "FromISR" API, like xQueueSendFromISR() so that the same functionality is achieved.

# FREERTOS HOOK FUNCTIONS

FreeRTOS hook (or callback) functions are user-defined functions that can be optionally provided to extend or customize the behavior of FreeRTOS. These functions are called by the FreeRTOS kernel at specific points in the execution. They provide a mechanism for developers to add application-specific functionality without modifying the core FreeRTOS code.

## Idle task Hook function

As previously mentioned, the idle task is a special task that runs whenever no other tasks are ready to run.

The Idle task hook function in FreeRTOS, typically represented as `vApplicationIdleHook( void );` is a user defined function that can be added to your application to execute custom code when the system is idle such as background processing or sending the MCU into low-power mode since idle task means no immediate processing required or tasks running.

In order to use the idle task hook function you must set the macro `configUSE_IDLE_HOOK` to 1 in your FreeRTOSConfig.h file. After setting this macro then you can define the in your application, like for example you can define this function in main.c

```
vApplicationIdleHook( void )
{
   //Do something
}
```

Then, the FreeRTOS kernel will call `vApplicationIdleHook( void );` during each cycle of the idle task

# Other FreeRTOS Hook Functions

Apart from the idle task hook function, there are many other hook functions:

**1. Tick Hook (vApplicationTickHook):**

- Called during each tick interrupt.
- Useful for periodic actions that need to be performed regularly.
- Enabled via the configUSE_TICK_HOOK configuration option.

**2. Malloc Failed Hook (vApplicationMallocFailedHook):**

- Called when a call to pvPortMalloc fails because there is insufficient free memory.
- Useful for logging memory allocation issues or triggering a system reset.
- Enabled via the configUSE_MALLOC_FAILED_HOOK configuration option.

**4. Stack Overflow Hook (vApplicationStackOverflowHook):**

- Called if a task stack overflows.
- The hook function can take actions like logging the issue or resetting the system.
- Requires stack overflow checking to be enabled (configCHECK_FOR_STACK_OVERFLOW).

**5. Daemon Task Startup Hook (vApplicationDaemonTaskStartupHook):**

- Called when the Daemon (or Timer service) task starts.
- Useful for initializing resources used by the Daemon task.
- Enabled via configUSE_DAEMON_TASK_STARTUP_HOOK.

**6. Get Idle Task Memory Hook (vApplicationGetIdleTaskMemory):**

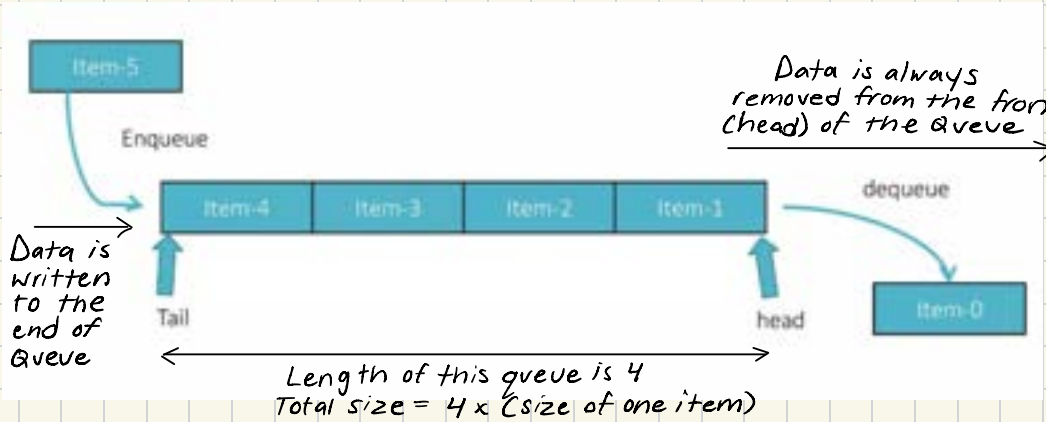
- Allows the application to provide the memory that is used by the Idle task.
- Useful in systems where memory allocation is static and not done via pvPortMalloc.

These hook functions can all, optionally, be implemented and the FreeRTOS kernel will call these hook functions whenever the corresponding events happen.

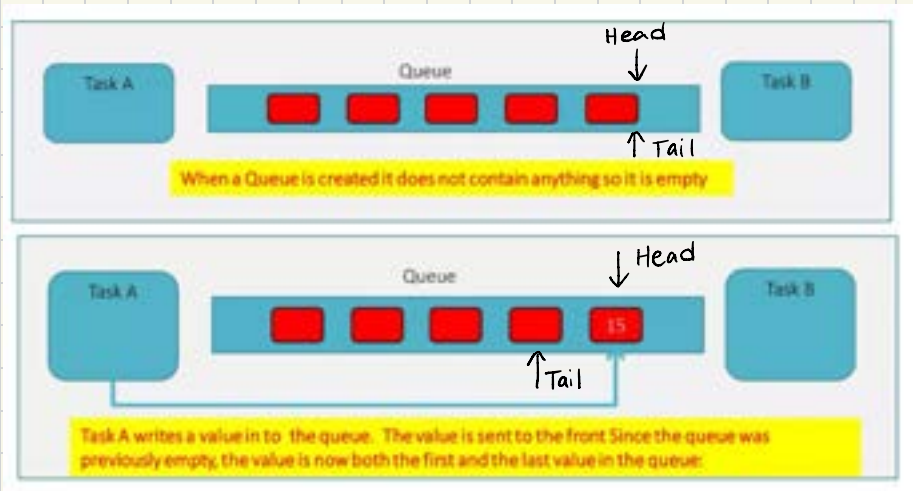
# FREERTOS QUEUE MANAGEMENT

## What are Queues?

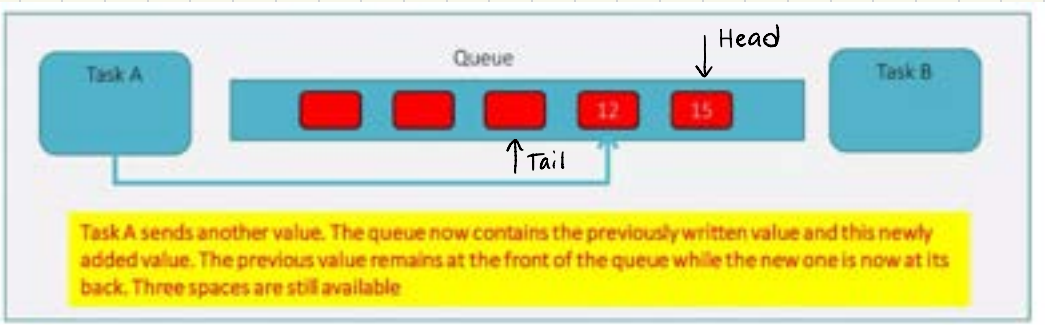
Queues in FreeRTOS are data structures used for inter-task communication, allowing tasks to send and receive messages safely and efficently. The queue data structure can hold a finite number of fixed size data items.

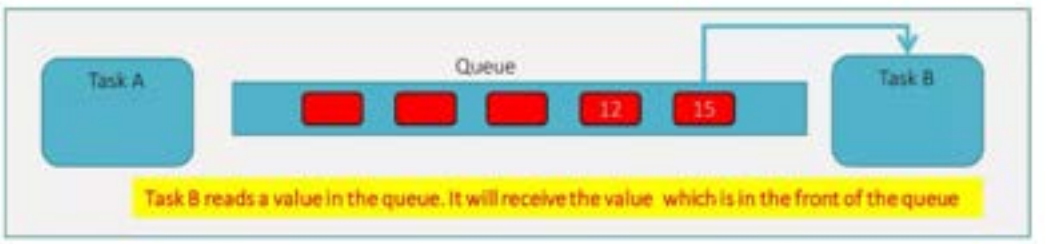


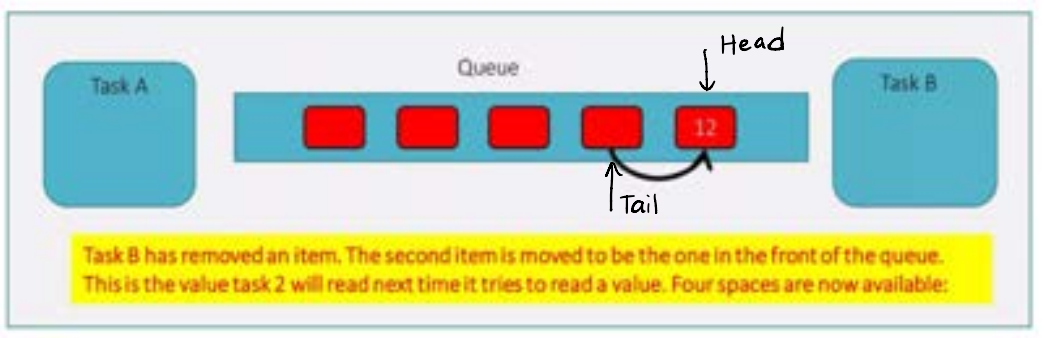

Queues operate on a First-In-First-Out basis. This means that the first message placed in the Queue is the first message to be removed.




Each item = 1 byte
So total size = 5 elements x 1 = 5 bytes




## Creating a Queue

FreeRTOS API to create a Queue:

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength,
                            UBaseType_t uxItemSize );
```

`UBaseType_t uxQueueLength` → How many items should the Queue hold? i.e. Length of Queue

`UBaseType_t uxItemSize` → What is the size of a single item in bytes

**returns: QueueHandle_t** → If the Queue is created successfully, then a handle to the created Queue, which is of type: QueueHandle_t, is returned. If the memory required to create the Queue could not be allocated and the Queue is not successfully created then NULL is returned. This handle, if the Queue creation is successful, is used to do lots of Queue operations, such as writing (enqueueing), reading (dequeueing), seeking, deleting, etc.

**Example usage:**

```
struct AMessage
{
    char ucMessageID;
    char ucData[ 20 ];
};

void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;

    /* Create a queue capable of containing 10 unsigned long values. */
    xQueue1 = xQueueCreate( 10, sizeof( unsigned long ) );

    if( xQueue1 == NULL )
    {
        /* Queue was not created and must not be used. */
    }

    /* Create a queue capable of containing 10 pointers to AMessage
    structures.  These are to be queued by pointers as they are
    relatively large structures. */
    xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

    if( xQueue2 == NULL )
    {
        /* Queue was not created and must not be used. */
    }

    /* ... Rest of task code. */
 }
```

## Sending data to a Queue

`xQueueSendToFront( )` → Sends data to the front (head) of Queue

`xQueueSendToBack( )` → Sends data to the back (tail) of Queue

## xQueueSendToFront()

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

`QueueHandle_t xQueue` → The Queue handle that was received by the QueueCreate API

`const void * pvItemToQueue` → The memory address of the data item which should be added to the Queue.

`TickType_t xTicksToWait` → The number of ticks to wait if the Queue is full. If you dont want to wait and want the function to immediately return saying that the Queue is full, then this value should be set to zero. If you set this value as the macro: `port_MAX_DELAY`, then the task that called this function will wait for the Queue to become free for atleast one item. If the Queue never becomes free, then the task will be waiting forever.

`returns: BaseType_t` → returns `pdTRUE` if the item was successfully posted, otherwise `errQUEUE_FULL`

Example usage:

```
struct AMessage
{
    char ucMessageID;
    char ucData[ 20 ];
} xMessage;

unsigned long ulVar = 10UL;

void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;
struct AMessage *pxMessage;

    /* Create a queue capable of containing 10 unsigned long values. */
    xQueue1 = xQueueCreate( 10, sizeof( unsigned long ) );

    /* Create a queue capable of containing 10 pointers to AMessage
    structures.  These should be passed by pointer as they contain a lot of
    data. */
    xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

    /* ... */

    if( xQueue1 != 0 )
    {
        /* Send an unsigned long.  Wait for 10 ticks for space to become
        available if necessary. */
        if( xQueueSendToFront( xQueue1,
                               ( void * ) &ulVar,
                               ( TickType_t ) 10 ) != pdPASS )
        {
            /* Failed to post the message, even after 10 ticks. */
        }
    }

    if( xQueue2 != 0 )
    {
        /* Send a pointer to a struct AMessage object.  Don't block if the
        queue is already full. */
        pxMessage = & xMessage;
        xQueueSendToFront( xQueue2, ( void * ) &pxMessage, ( TickType_t ) 0 );
    }

    /* ... Rest of task code. */
}
```

## xQueueSendToBack()

Same as xQueueSendToFront() except that instead of sending the

item to the front (head), it will be added to the back (tail) of the Queue.

## Receiving data from a Queue

`xQueueReceive( )`

`xQueuePeek( )`

## xQueueReceive( )

This API will read a data item from the Queue. The data item which is being read will be removed from the Queue.

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

`QueueHandle_t xQueue` → The Queue handle that was received by the QueueCreate API

`void *pvBuffer` → The pointer which you supply here will hold the data item which is being read.

`TickType_t xTicksToWait` → The number of ticks to wait if the Queue is empty. If you dont want to wait and want the function to immediately return saying that the Queue is empty, then this value should be set to zero. If you set this value as the macro: `port_MAX_DELAY`, then the task that called this function will wait for the Queue to become occupied atleast one item. If the Queue never becomes occupied then the task will be waiting forever.

`returns: BaseType_t` → returns `pdTRUE` if the item was successfully received from the Queue, otherwise `pdFALSE`

xQueuePeek()

This API is exactly the same as xQueueReceive() but it does not remove the item which is being read from the Queue.

# FREERTOS SOFTWARE TIMERS

Lets say that we have a function: ToggleLED(), and we want to call this function every 500 ms. Generally, we have two ways to implement this:

## Hardware Timers

- Handled by the TIMER peripheral of the MCU
- No FreeRTOS API's. You have to create your own function to initialize and manage the TIMER peripherals
- µs/ns resolutions can be achieved.

## Software Timers

- Handled by the FreeRTOS Kernel
- FreeRTOS API's are available
- Resolution depends on the Macro: `RTOS_TICK_RATE_HZ`

  Which can be found in the "FreeRTOSConfig.h" file

Software Timers API

# Software Timers

[API]

## FreeRTOS Software Timer API Functions

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

This information can be found from the FreeRTOS website.

xTimerCreate()

# xTimerCreate

[Timer API]

timers.h

```
TimerHandle_t xTimerCreate
                ( const char * const pcTimerName,
                  const TickType_t xTimerPeriod,
                  const UBaseType_t uxAutoReload,
                  void * const pvTimerID,
                  TimerCallbackFunction_t pxCallbackFunction );
```

Creates a new software timer instance and returns a handle by which the timer can be referenced.

For this RTOS API function to be available:

1. configUSE_TIMERS and configSUPPORT_DYNAMIC_ALLOCATION must both be set to 1 in FreeRTOSConfig.h (configSUPPORT_DYNAMIC_ALLOCATION can also be left undefined, in which case it will default to 1).
2. The FreeRTOS/Source/timers.c C source file must be included in the build.

Each software timer requires a small amount of RAM that is used to hold the timer's state. If a timer is created using xTimerCreate() then this RAM is automatically allocated from the FreeRTOS heap. If a software timer is created using xTimerCreateStatic() then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time. See the Static Vs Dynamic allocation page for more information.

Timers are created in the dormant state. The xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod() and xTimerChangePeriodFromISR() API functions can all be used to transition a timer into the active state.

**Parameters:**

| | |
|---|---|
| *pcTimerName* | A human readable text name that is assigned to the timer. This is done purely to assist debugging. The RTOS kernel itself only ever references a timer by its handle, and never by its name. |
| *xTimerPeriod* | The period of the timer. The period is specified in ticks, and the macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds to a time specified in ticks. For example, if the timer must expire after 100 ticks, then simply set xTimerPeriod to 100. Alternatively, if the timer must expire after 500ms, then set xTimerPeriod to pdMS_TO_TICKS( 500 ). pdMS_TO_TICKS() can only be used if configTICK_RATE_HZ is less than or equal to 1000. The timer period must be greater than 0. |
| *uxAutoReload* | If uxAutoReload is set to pdTRUE, then the timer will expire repeatedly with a frequency set by the xTimerPeriod parameter. If uxAutoReload is set to pdFALSE, then the timer will be a one-shot and enter the dormant state after it expires. |
| *pvTimerID* | An identifier that is assigned to the timer being created. Typically this would be used in the timer callback function to identify which timer expired when the same callback function is assigned to more than one timer, or together with the vTimerSetTimerID() and pvTimerGetTimerID() API functions to save a value between calls to the timer's callback function. |
| *pxCallbackFunction* | The function to call when the timer expires. Callback functions must have the prototype defined by TimerCallbackFunction_t, which is:<br>`void vCallbackFunction( TimerHandle_t xTimer );`<br>. |

**Returns:**

If the timer is created successfully then a handle to the newly created timer is returned. If the timer cannot be created because there is insufficient FreeRTOS heap remaining to allocate the timer structures then NULL is returned.

Example usage:

```
#define NUM_TIMERS 5

/* An array to hold handles to the created timers. */
TimerHandle_t xTimers[ NUM_TIMERS ];

/* Define a callback function that will be used by multiple timer
instances.  The callback function does nothing but count the number
of times the associated timer expires, and stop the timer once the
timer has expired 10 times.  The count is saved as the ID of the
timer. */
void vTimerCallback( TimerHandle_t xTimer )
{
const uint32_t ulMaxExpiryCountBeforeStopping = 10;
uint32_t ulCount;

   /* Optionally do something if the pxTimer parameter is NULL. */
   configASSERT( xTimer );

   /* The number of times this timer has expired is saved as the
   timer's ID.  Obtain the count. */
   ulCount = ( uint32_t ) pvTimerGetTimerID( xTimer );

   /* Increment the count, then test to see if the timer has expired
   ulMaxExpiryCountBeforeStopping yet. */
   ulCount++;

   /* If the timer has expired 10 times then stop it from running. */
   if( ulCount >= ulMaxExpiryCountBeforeStopping )
   {
       /* Do not use a block time if calling a timer API function
       from a timer callback function, as doing so could cause a
       deadlock! */
       xTimerStop( xTimer, 0 );
   }
   else
   {
      /* Store the incremented count back into the timer's ID field
      so it can be read back again the next time this software timer
      expires. */
      vTimerSetTimerID( xTimer, ( void * ) ulCount );
   }
}

void main( void )
{
long x;

    /* Create then start some timers.  Starting the timers before
    the RTOS scheduler has been started means the timers will start
    running immediately that the RTOS scheduler starts. */
    for( x = 0; x < NUM_TIMERS; x++ )
    {
        xTimers[ x ] = xTimerCreate
                  ( /* Just a text name, not used by the RTOS
                    kernel. */
                    "Timer",
                    /* The timer period in ticks, must be
                    greater than 0. */
                    ( 100 * x ) + 100,
```

```
                    /* The timers will auto-reload themselves
                    when they expire. */
                    pdTRUE,
                    /* The ID is used to store a count of the
                    number of times the timer has expired, which
                    is initialised to 0. */
                    ( void * ) 0,
                    /* Each timer calls the same callback when
                    it expires. */
                    vTimerCallback
                  );

        if( xTimers[ x ] == NULL )
        {
            /* The timer was not created. */
        }
        else
        {
            /* Start the timer.  No block time is specified, and
            even if one was it would be ignored because the RTOS
            scheduler has not yet been started. */
            if( xTimerStart( xTimers[ x ], 0 ) != pdPASS )
            {
                /* The timer could not be set into the Active
                state. */
            }
        }
    }

    /* ...
    Create tasks here.
    ... */

    /* Starting the RTOS scheduler will start the timers running
    as they have already been set into the active state. */
    vTaskStartScheduler();

    /* Should not reach here. */
    for( ;; );
}
```

xTimerStart()

# xTimerStart

[Timer API]

timers.h

```
BaseType_t xTimerStart( TimerHandle_t xTimer,
                        TickType_t xBlockTime );
```

Software timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the RTOS kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.

xTimerStart() starts a timer that was previously created using the xTimerCreate() API function. If the timer had already been started and was already in the active state, then xTimerStart() has equivalent functionality to the xTimerReset() API function.

Starting a timer ensures the timer is in the active state. If the timer is not stopped, deleted, or reset in the mean time, the callback function associated with the timer will get called 'n 'ticks after xTimerStart() was called, where 'n' is the timers defined period.

It is valid to call xTimerStart() before the RTOS scheduler has been started, but when this is done the timer will not actually start until the RTOS scheduler is started, and the timers expiry time will be relative to when the RTOS scheduler is started, not relative to when xTimerStart() was called.

The configUSE_TIMERS configuration constant must be set to 1 for xTimerStart() to be available.

**Parameters:**

| | |
|---|---|
| *xTimer* | The handle of the timer being started/restarted. |
| *xBlockTime* | Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the start command to be successfully sent to the timer command queue, should the queue already be full when xTimerStart() was called. xBlockTime is ignored if xTimerStart() is called before the RTOS scheduler is started. |

**Returns:**
pdFAIL will be returned if the start command could not be sent to the timer command queue even after xBlockTime ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when xTimerStart() is actually called. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.
**Example usage:**
See the example on the xTimerCreate() documentation page.

# SEMAPHORES

## What is Synchronization?

Synchronization is the aligning of tasks to achieve optimal behavior

## Why is Synchronization important?

For example, lets say Task A is continuously checking to see if temperature data is available from a temperature sensor. Now, lets say Task B checks in every 5ms to see if Task A has any new data. As this happens, lets say Task B checks in with Task A 100 times, and only 2 times Task A had any data for Task B. This can be called as "un-synchronized". Task B unnecessarily checks in 98 times, wasting precious CPU clock cycles.

But, now lets say that instead of Task B checking in with Task A every 5ms, Task A simply signals Task B whenever there is new data available, thus only calling Task B when needed, this approach saves CPU cycles. This can be called as "synchronized"

## What is Mutual Exclusion?

Mutual exclusion is making sure that if there is a resource that is shared between multiple tasks, that resource should only be accessed by one task at a time.

## Why is Mutual Exclusion Important?

For example, lets say that we have a global variable in our application, char *msg= "Hello". Now, Task A's job is to read that value and print it to the terminal. As Task A is reading that value, another task suddenly changes the last 2 characters of the string to 'FF'. Now, when Task A finishes reading the string, instead of "Hello", it read "HelIFF" and prints that instead.

Now, lets say that we get a Lock to the "Hello" string, this way only one task can have the lock, so task A will get the lock, While it has the lock, Task B cannot access the string. So Task A will read the string without any interruptions and when its done, it can give the lock to Task B.

## What are Semaphores?

**A semaphore in an RTOS is like a flag that is used to signal the availability or status of a resource. It helps manage access to that resource among various tasks or threads. Think of it as a key to a shared locker. Only the task with the key can access the locker at any given time. When a task is done, it hands over the key to the next task in line.**

**Now, let's apply this concept to a theoretical RTOS application:**

**Context:**
**We have a smart home system with two key tasks:**

1. **Sensor Data Processing Task: Reads data from sensors (temperature, motion).**
2. **Data Logging Task: Logs this sensor data to storage or a cloud server.**

**Both tasks need to access a shared resource – the sensor data buffer.**

**Using a Semaphore for Mutual Exclusion:**

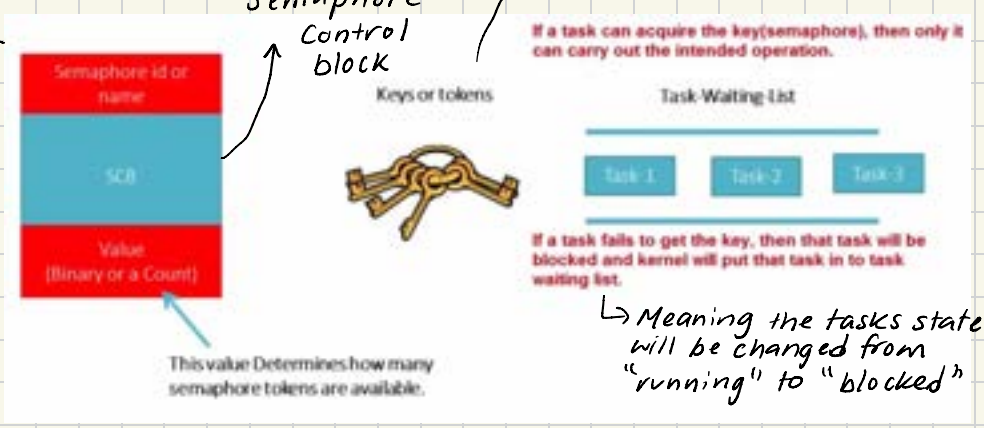
- Imagine the sensor data buffer as a locker.
- The Semaphore acts as the key to this locker.
- When the Sensor Data Processing Task is collecting and storing data, it holds the key (takes the semaphore). This ensures no other task can access the buffer and potentially corrupt the data.
- After it's done, it hands over the key (releases the semaphore), making the buffer available for the Data Logging Task.

**Using a Semaphore for Synchronization:**

- For synchronization, the semaphore serves as a signal flag.
- Once the Sensor Data Processing Task has new data in the buffer, it raises the flag (releases the semaphore).
- The Data Logging Task, waiting for this signal, sees the flag and starts logging the new data. This ensures the Data Logging Task operates only when there's new data to process.

**In this scenario, the semaphore effectively manages access to the shared data buffer and coordinates the operation of both tasks, ensuring they work in harmony without interfering with each other.**

## Creation of a Semaphore




This is whats created when a "semaphore" is created

A single semaphore can be aquired finite number of times by the tasks depending upon how the semaphore is first initialized.

The semaphore "value" which can be of type binary (1 or 0) or type count (normal number) determines how many semaphore tokens are available.

For example, If a semaphore's value = 4, then 4 semaphore keys or tokens are available to be acquired. If a task tries to acquire the key for the 5$^{th}$ time, then it will be blocked.

This value (4) decreases whenever the key is acquired and it increases when the semaphore token or key is given back.

When the semaphore value reaches zero, this means that all the available tokens or keys have been acquired and there are none left. At this time, if any task tries to take the semaphore, then the task will be blocked and moved to the waiting list.

Blocked tasks are kept in the task waiting list, and if pre-emptive scheduling is enabled, the tasks will be in order of priority. This means that if multiple blocked tasks are waiting for a semaphore, if that semaphore becomes available, the highest priority task in the waiting list will get the freshly available semaphore and thus come out of the blocked state and out of the waiting list.

Types of semaphores

1. Binary Semaphore
2. Counting Semaphore

## 1. Binary Semaphore

As the name indicates, this semaphore only works on 2 values; 1 or 0


If the value = 1, then the semaphore or key is available

If the valve = 0, then the key is not available

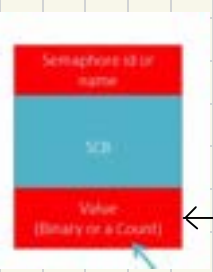
This "valve" can be either 1 or 0

## Binary Semaphore Use cases

1. Synchronization
That is synchronization between tasks or synchronization between interrupts and tasks.

2. Mutual Exclusion
A Binary semaphore can also be used for Mutual Exclusion, that is to guard a critical section of code.

## 2. Counting Semaphore




This "valve" can be any number. If you initialize it to 5 for example, then it is analogous to having 5 keys

# Counting Semaphore Use Cases

## 1. Counting Events

In this usage scenario, an event handler will "give" a semaphore each time an event occurs - causing the semaphore's count value to be incremented on each give. A handler task will "take" the a semaphore each time it processes an event - causing the semaphore's count to decrease.

Example usage:

```
void interrupt_handler(void)
{
    do_important_work(); /* this is very short code */
    sema_give(&sem);     /* Give means release the key */
}/* exit from the interrupt */

void task_function(void)
{
    /* if taking a key is un-successful then this task will be blocked until key is available */
    while ( sem_get(&sem) ) // GET means, trying to take the key
    {
        // it will come here, only if taking a key is successful..
        /* Do time consuming work of the ISR */
    }
}
```

Each time an event/interrupt occurs, the event/interrupt handler will "give" the semaphore, thus increasing the count value. Thus it can be that the events/interrupts are being logged

A handler task which was previously blocked waiting for a free semaphore key, now unblocks as the semaphore becomes available and handles all the events or interrupts that are logged by "taking" the semaphore

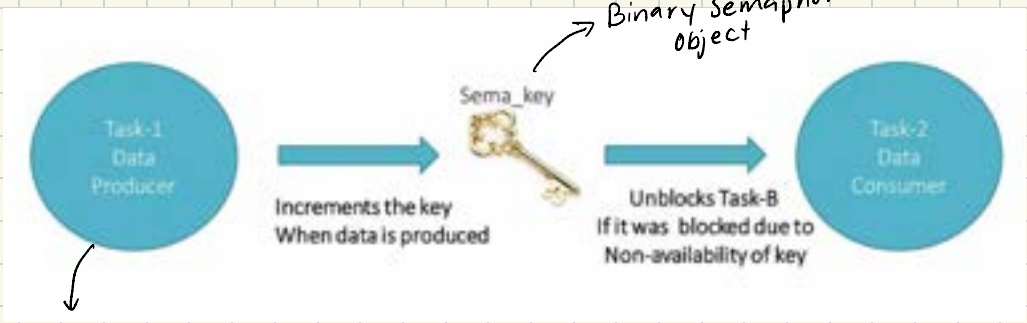
## 2. Resource Management

In this usage scenario, the count value indicates the number of resources available. To obtain, control of a resource, a task must first obtain a semaphore by decrementing the semaphore's count value. When the count value reaches zero, there are no available resources. When a task finishes with a resource, it "gives" the semaphore back, thus incrementing the semaphore's count value.

# Binary Semaphore to achieve Synchronization between two tasks

The main use of a binary semaphore or counting semaphore is synchronization. The synchronization can be either between two tasks or between a task and an interrupt.

For example, synchronization between two tasks:




When data is produced, Task-1 will increment the semaphore value

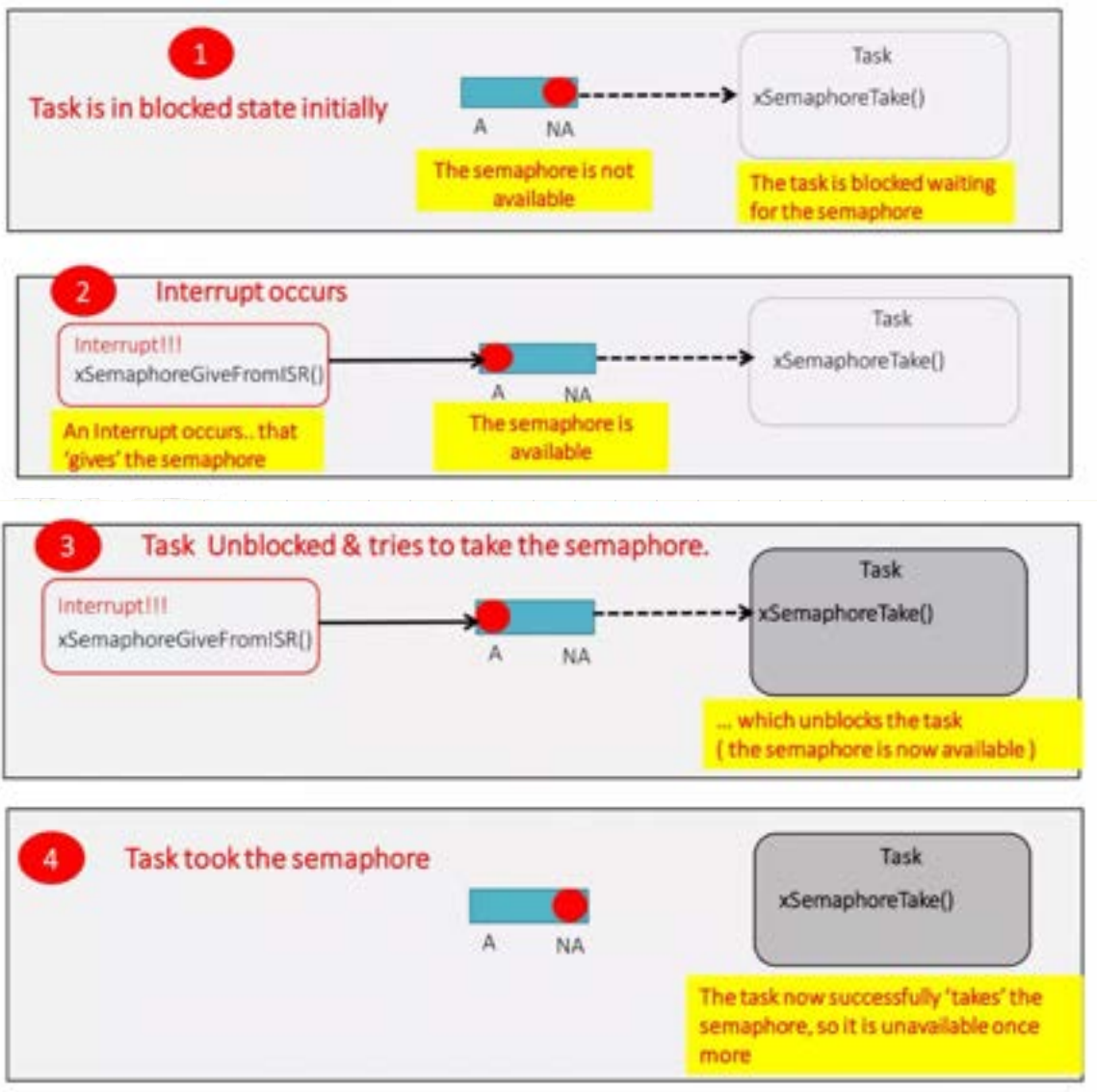
```
void task1_running(void)
{
    if( TRUE == produce_some_data())
    {
        /* This is a signalling for the task2  to wake up if it is blocked due to non availability of key */
        sema_give(&sema_key); // 'GIVE' operation will increment the semaphore value by 1
    }
}
void task2_running(void)
{
    /* if sema_key is unavailable then task2 will be blocked. */
    /*If sema_key is available, then task2 will take it and sema_key becomes unavailable again
    */
      while( sema_take(&sema_key) )
      {
        //Task will come here only when the sem_take operation is successful.
        /* lets consume the data */
        /* since the sem_key value is zero at this point, the next time when task-2 tries to
    take, it will be blocked. */
      }
}
```

Once Task-1 gives the sema_key this task will be unblocked because of this line

Binary Semaphore to achieve synchronization between task and interrupt.

The binary semaphore is very handy and well suited to achieve synchronization between an interrupt handler and a Task handler.

For example:



1. An interrupt occured, when the task was in blocked state.

2. The ISR executed and gave the semaphore, dve to that, the task was unblocked.

3. The task executed and took the semaphore

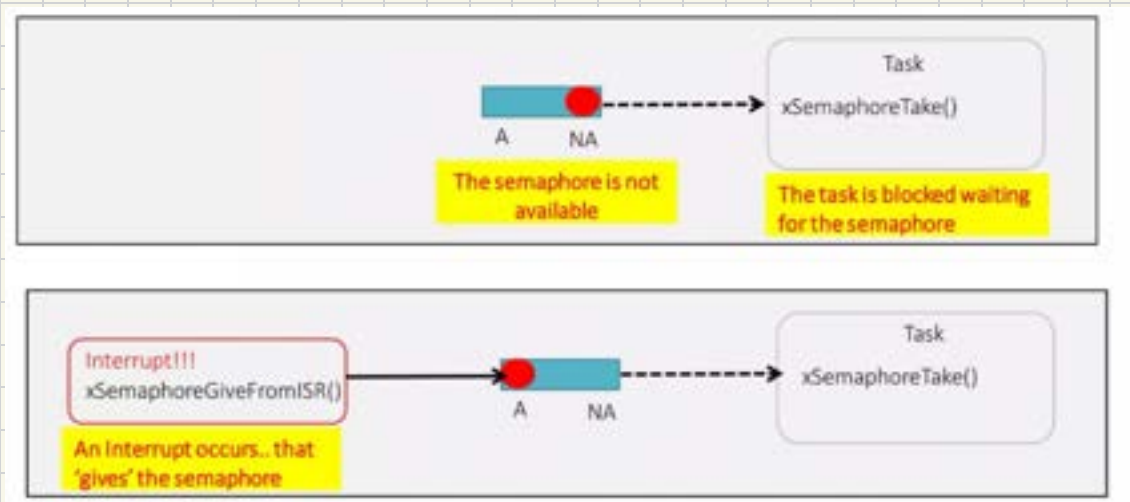
4. The task performed its intended work and tried to take the semaphore once again.

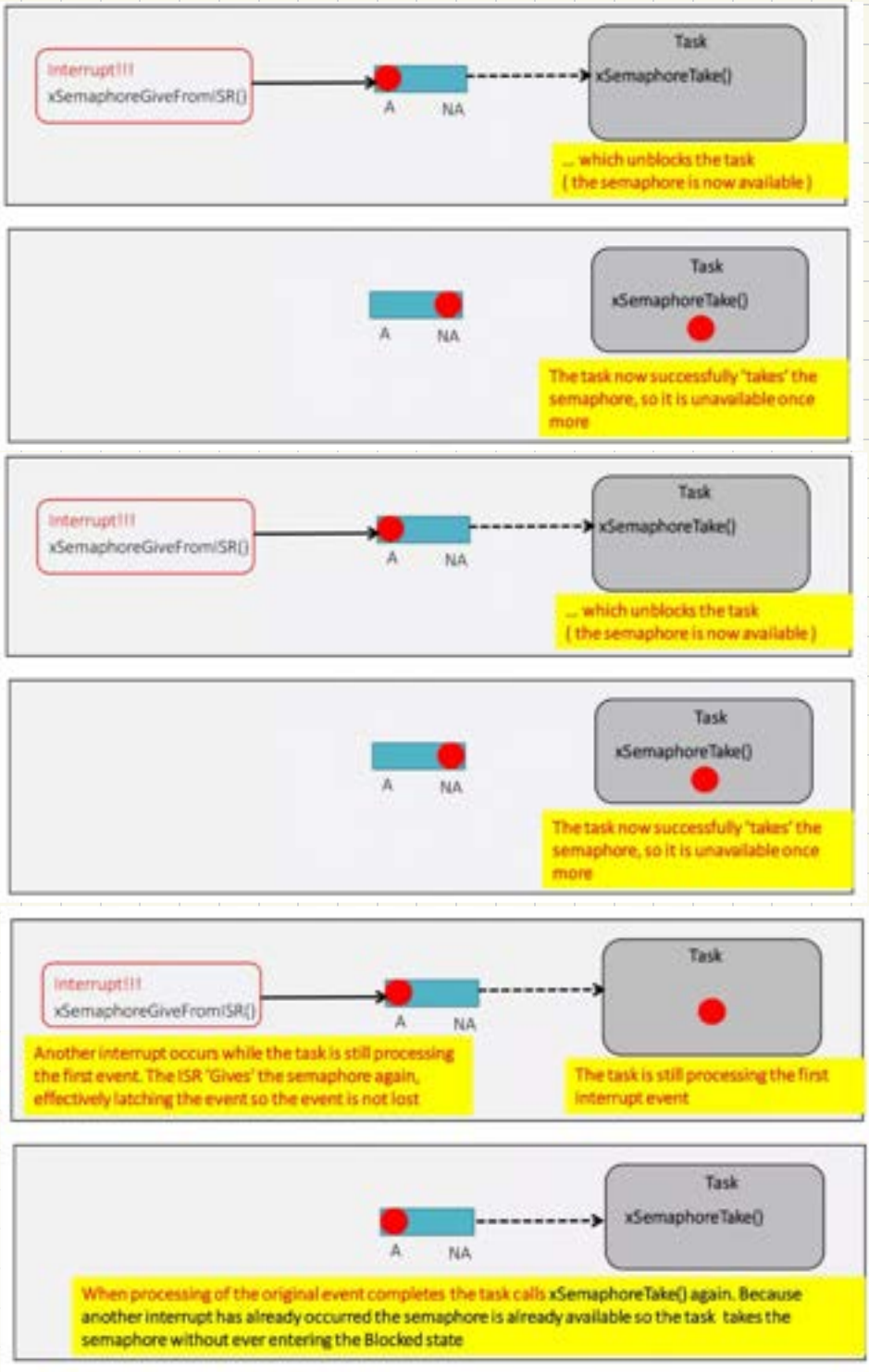
5. Since the semaphore is not available, the task enters the blocked state, waiting for the next "give" from the next ISR execution.

## Events Latching and Processing using Binary Semaphores

Latching is when a semaphore is given to a task that has just recieved a semaphore.

For example:




Interrupt!!!
xSemaphoreGiveFromISR()
A
NA
Task
xSemaphoreTake()
... which unblocks the task
( the semaphore is now available )
A
NA
Task
xSemaphoreTake()
The task now successfully 'takes' the semaphore, so it is unavailable once more
Interrupt!!!
xSemaphoreGiveFromISR()
A
NA
Task
xSemaphoreTake()
... which unblocks the task
( the semaphore is now available )
A
NA
Task
xSemaphoreTake()
The task now successfully 'takes' the semaphore, so it is unavailable once more
Interrupt!!!
xSemaphoreGiveFromISR()
A
NA
Task
Another interrupt occurs while the task is still processing the first event. The ISR 'Gives' the semaphore again, effectively latching the event so the event is not lost
The task is still processing the first interrupt event
A
NA
Task
xSemaphoreTake()
When processing of the original event completes the task calls xSemaphoreTake() again. Because another interrupt has already occurred the semaphore is already available so the task takes the semaphore without ever entering the Blocked state

## Conclusion

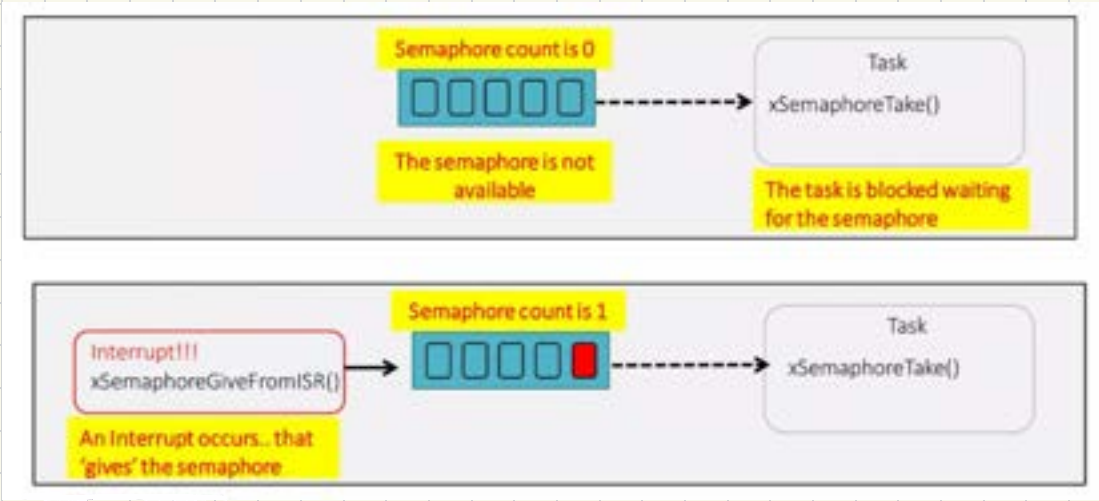
1. When interrupts/events happen relatively slow, the binary semaphore can latch at most only one event.

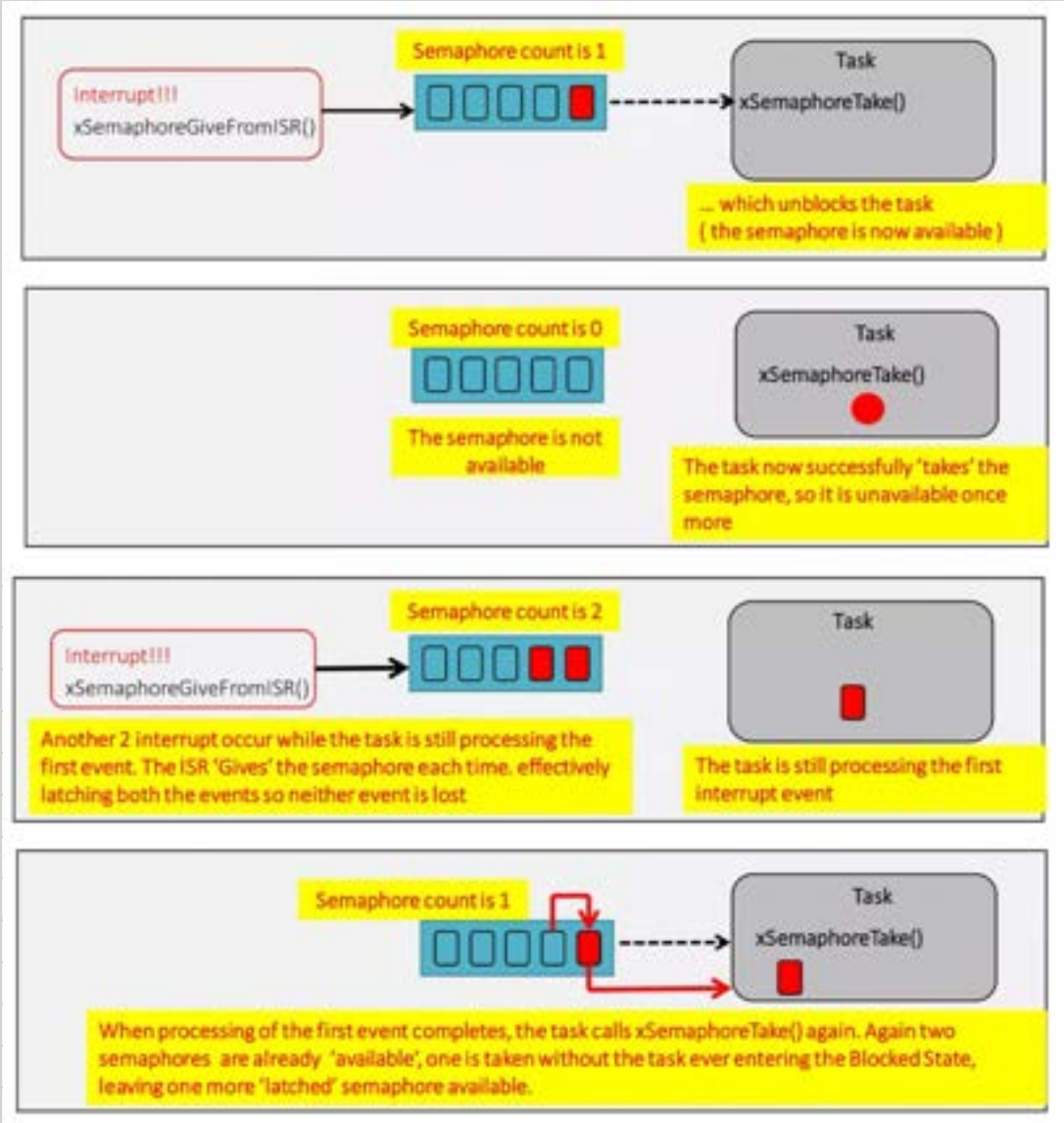
2. If multiple interrupts/events trigger back to back, then the binary semaphore will not be able to latch all the events. So some events will be lost.

3. How can we solve this issue? Introducing... Counting Semaphores!

## Counting Semaphore to latch and process Multiple events

Lets say that we create a counting semaphore with valve 5.






Conclusion

You can use counting semaphores to count the events and then process them serially one by one using another task.

The counting semaphore can also be used for resource management. That is to regulate access to multiple identical resources.

# MUTUAL EXCLUSION

Mutual exclusion is a concept used to prevent multiple proccesses or threads from accessing a shared resource, like data or hardware, at the same time. This is crucial to prevent data corruption or inconsistent states. When a process or task accesses a resource, it "locks" it, performs the necessary operations, and then "unlocks" it, allowing other processes or tasks to access the resource in a controlled and safe manner. Mutual exclusion can be implemented in FreeRTOS using Binary Semaphores or Mutexes.

## Mutual Exclusion using Binary Semaphore

Access to a resource that is shared either between tasks or between tasks and interrupts, need to be serialized using some techniques to ensure data consistency.

Usually a common code block which deals with global arrays, variables, or memory addresses, has the potential to get corrupted when many tasks or interrupts are racing around it trying to access it at the same time.

There are two ways to Implement Mutual Exclusion in FreeRTOS,
1. Using Binary Semaphore API
2. Using Mutex APIs

The previous section explained how binary Semaphores can be used for Synchronization. The same binary Semaphore can also be used to avoid race conditions or you can say to achieve the mutual exclusion.

For example:

Thread Unsafe Code:

global variable

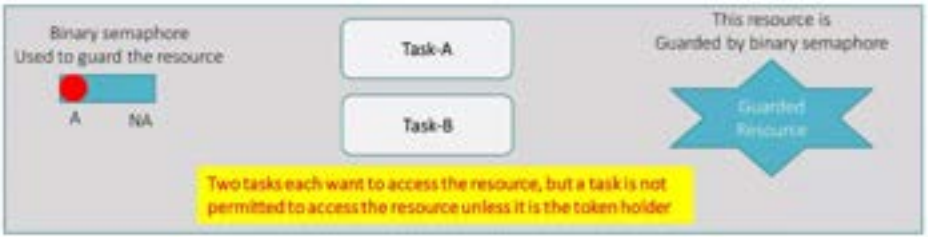
```
#define  UART_DR              *((unsigned long * ) (0x40000000) )
/* This is a common function which write to UART DR */
int UART_Write( uint32_t len , uint8_t *buffer)
{
    for (uint32_t i=0;i < len ; i++)
        {
        /*  if Data Register is empty write it  */
           while(! is_DR_empty() );
            UART_DR = buffer[i];
        }
}
```

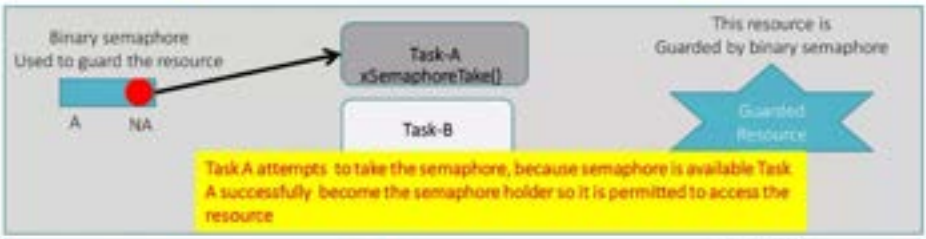
We are accessing global data but any other task can also access the data and it could change the data as we access it. This could result in data corruption, unpredictable behavior and system crashes

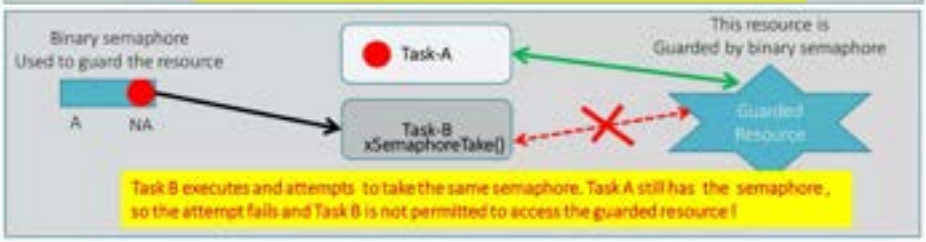
## Mutual Exclusion using Binary Semaphores

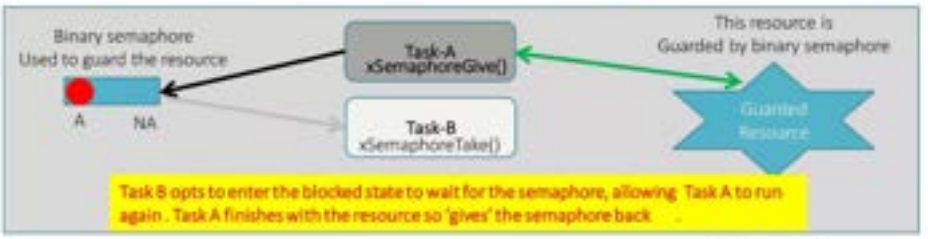
```
#define  UART_DR              *((unsigned long * ) (0x40000000) )
/* This is a common function which write to UART DR */
int UART_Write( uint32_t len , uint8_t *buffer)
{
    for (uint32_t i=0;i < len ; i++)
        {
          sema_take_key( bin_sema );
        /*  if Data Register is empty write it  */
           while(! is_DR_empty() );
            UART_DR = buffer[i]; //Critical section
         sema_give_key( bin_sema );
        }
}
```

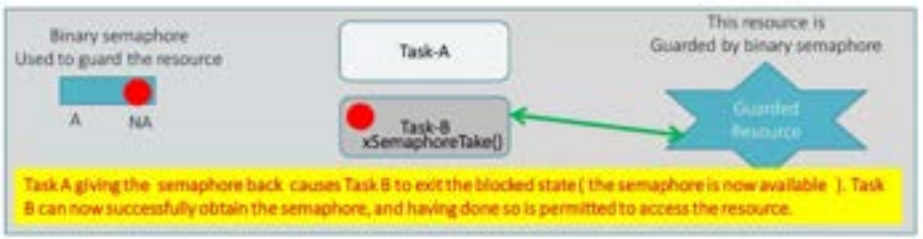
In this case, if TaskA executes this code and takes the semaphore, which makes the binary semaphore have a value of 0. Now, if any other task tries to execute the same code, when it reaches the "take key" step, it will remain in the blocked state since the semaphore value is zero. Only when Task A finishes the changes to the critical code and executes the "give key" step, then the semaphore value will go back to 1, and the other task will unblock, take the key and then do its work

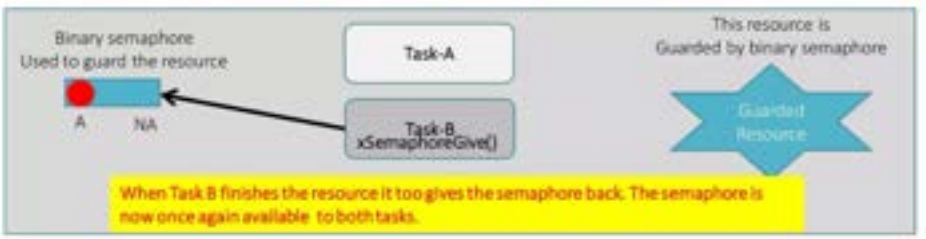

Binary semaphore
Used to guard the resource
A
NA
Task-A
Task-B
This resource is
Guarded by binary semaphore
Guarded Resource
Two tasks each want to access the resource, but a task is not permitted to access the resource unless it is the token holder


Binary semaphore
Used to guard the resource
A
NA
Task-A
xSemaphoreTake()
Task-B
This resource is
Guarded by binary semaphore
Guarded Resource
Task A attempts to take the semaphore, because semaphore is available Task A successfully become the semaphore holder so it is permitted to access the resource


Binary semaphore
Used to guard the resource
A
NA
Task-A
Task-B
xSemaphoreTake()
This resource is
Guarded by binary semaphore
Guarded Resource
Task B executes and attempts to take the same semaphore. Task A still has the semaphore , so the attempt fails and Task B is not permitted to access the guarded resource !


Binary semaphore
Used to guard the resource
A
NA
Task-A
xSemaphoreGive()
Task-B
xSemaphoreTake()
This resource is
Guarded by binary semaphore
Guarded Resource
Task B opts to enter the blocked state to wait for the semaphore, allowing Task A to run again . Task A finishes with the resource so 'gives' the semaphore back .


Binary semaphore
Used to guard the resource
A
NA
Task-A
Task-B
xSemaphoreTake()
This resource is
Guarded by binary semaphore
Guarded Resource
Task A giving the semaphore back causes Task B to exit the blocked state ( the semaphore is now available ). Task B can now successfully obtain the semaphore, and having done so is permitted to access the resource.


Binary semaphore
Used to guard the resource
A
NA
Task-A
Task-B
xSemaphoreGive()
This resource is
Guarded by binary semaphore
Guarded Resource
When Task B finishes the resource it too gives the semaphore back. The semaphore is now once again available to both tasks.

## Mutex Intro

A mutex is also a kind of binary semaphore that has a built in mechanism which minimizes the effect/potential for priority inversion that a normal binary semaphore does not have.

## What is Priority Inversion?

**Definition and Explanation of Priority Inversion**

**Straightforward Definition**

**Priority inversion is a scenario in real-time computing where a lower-priority task holds a shared resource needed by a higher-priority task. This situation causes the higher-priority task to wait unexpectedly, leading to performance issues and potentially missing critical deadlines.**

**Why It Occurs**

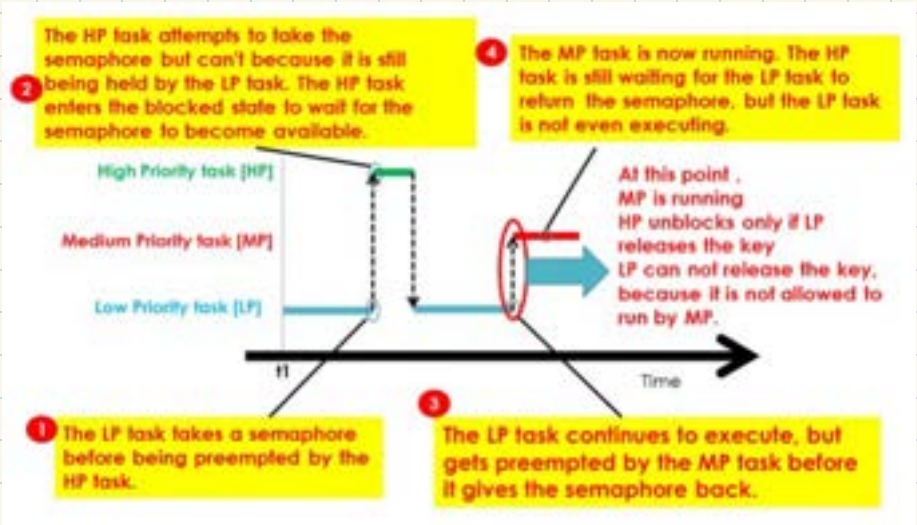
1. **Mixed Priority Tasks: Involves tasks of different priorities accessing shared resources.**
2. **Resource Locking: A low-priority task acquires a lock on a resource.**
3. **Preemption by Medium-Priority Task: A medium-priority task, which doesn't need the locked resource, preempts the low-priority task.**
4. **Blocking of High-Priority Task: The high-priority task, needing the same resource locked by the low-priority task, is blocked until the resource is released.**

**Impact of Priority Inversion**

1. **Missed Deadlines: High-priority tasks may miss their deadlines, leading to system performance degradation.**
2. **Reduced System Predictability: The system's behavior becomes less predictable due to the unexpected waiting times of high-priority tasks.**
3. **Potential System Failures: In critical systems, prolonged priority inversion can lead to failures.**

Generally, binary semaphores are the better choice for implementing synchronization (between tasks or between tasks and an interrupt), mutexes are the better choice for implementing mutual exclusion.

Example of Priority Inversion:




**Here , higher priority task is blocked and waiting for semaphore to gets released by lower priority task.But lower priority task is not allowed to run by the medium priority task until it finishes and exits.**

**As a result your lower Priority task may be delayed indefinitely and the higher priority task will remain in blocked state for indefinite time. This is exactly the scenario of Priority inversion. That means, even though the Task-2 has the highest priority the situation made it behave like lower Priority.**

**Remember that, until medium task appears everything is OK, but the moment medium priority task appears, then it may lead to priority inversion. So, as your application contains more tasks of different priority levels, then you have to test your application against priority inversion. There could be chances of higher priority tasks getting delayed unnecessarily.**

**So, are there any ways to solve or reduce the priority inversion problem ? Simple answer is don't use binary semaphore for mutual exclusion when your system has many tasks of different Priorities. Use MUTEX instead. MUTEX has inbuilt intelligence to minimize the priority inversion effect and this is the biggest difference between MUTEX and Binary semaphore.**

**MUTEX is also another kernel service, which is carefully designed to take care of this problem of priority inversion. Even though there are lots of similarities between mutex and binary semaphore, the mutex has inbuilt intelligence to reduce the problem like priority inversion.**

# Advantages of Mutex over Binary Semaphore

## Priority Inheritance

Mutexes and binary semaphores are very similar - the only major difference is that mutexes automatically provide a basic "Priority Inheritance" mechanism.

Priority Inheritance is a technique by which the mutex minimizes the negative effect of priority Inversion. A Mutex may not be able to fix the priority inversion problem completely but it surely lessens its impact.

Most of the RTOS, including FreeRTOS, have mutexes that implement a priority inheritance feature.

However, since mutexes have all these features to avoid priority inversion, the memory consumed by mutexes may be higher than a binary semaphore.

Example of Priority Inheritance: