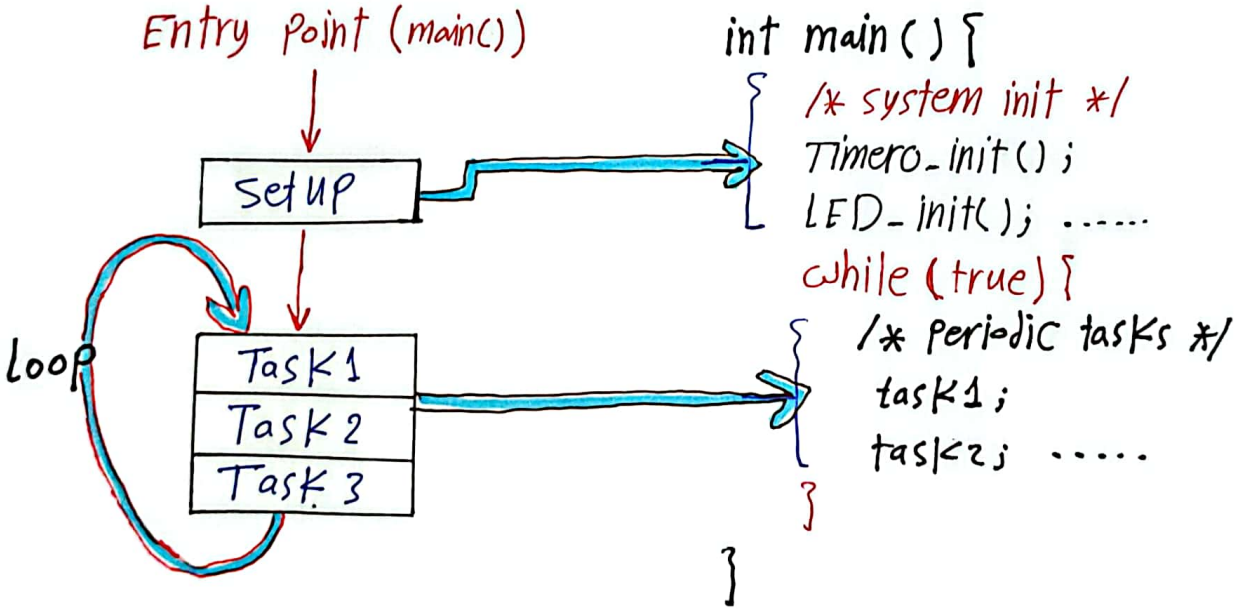
RTOS
(Real Time operating system)

* Embedded software Dynamic Architecture
→ super loop vs. Multi-threaded Application.

* super loop Architecture (also known as "Bare-Metal Programming"):-
→ the program executes within a single loop, typically referred to as the super loop or main loop
→ a super loop is a single thread that processes multiple tasks in a sequential order.
→ the thread continuously executes a loop that executes pending tasks one after another



→ in main() function, we set up any local variables, system initialization. and then perform one or more periodic tasks in a while (true) loop.
→ the super loop approach is suitable for applications that are not time critical and where the response time is not so important.
→ the super loop architecture is designed to avoid making blocking calls, which could halt program execution until a certain condition is met.
→ blocking calls, such as waiting for input or waiting for a resource to become available, can disrupt the sequential execution of tasks in the super loop.
→ By utilizing polling instead of blocking calls, the super loop can continuously check the status of resources without getting stuck or delaying other tasks

→ Polling allows the super loop to maintain responsiveness and keep the program flow active.

→ Based on the polling result, the super loop can decide on the appropriate actions to take or tasks to execute.

Determinism: knowing what happens at each point in time line.

Responsiveness: How fast I can respond to external events.

* **Advantages**:-
  - → very simple to use, edit and debug
  - → straight forward design
  - → Highly determinism
  - → Reduced memory footprint
  - → simple in S.W and minimum H.W resources are required.
* **Disadvantages**:-
  - → Low responsiveness
  - → High power consumption ⟹ it can't be used in battery powered embedded applications.
  - → None real time in execution behavior

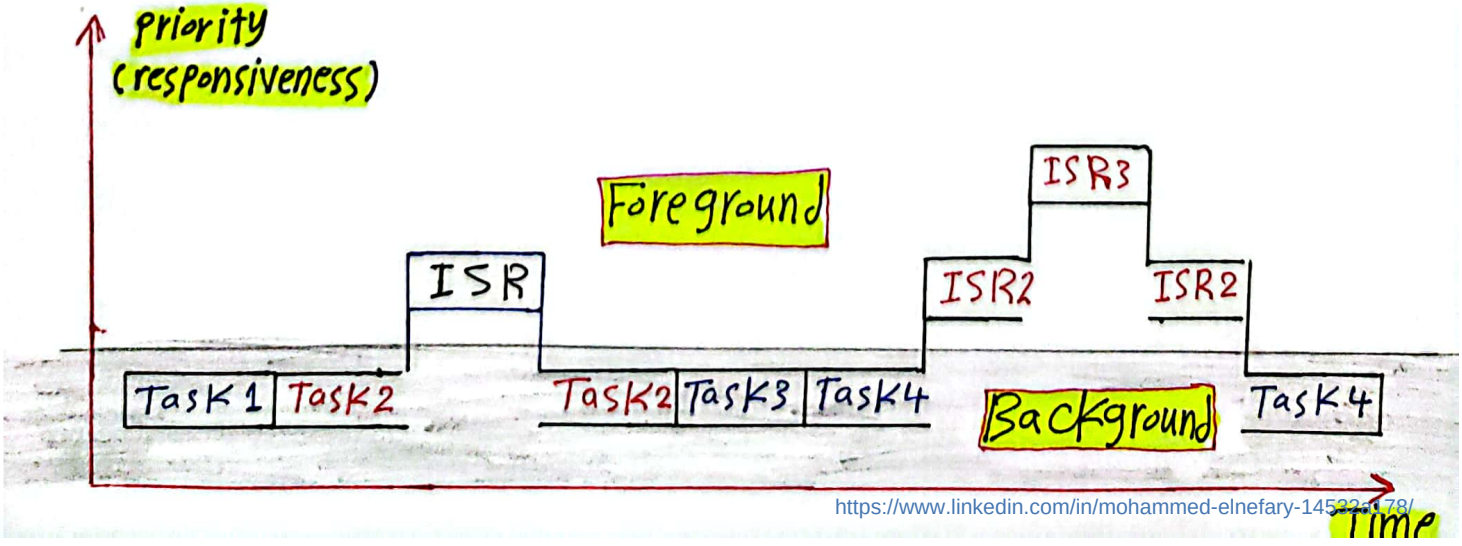
→ **Foreground / Background Architecture** :- (superloop / main + ISR)

→ Background ⟶ superloop

- the background is an infinite loop, typically inside the main function, that sequentially calls one or more functions.
- the background loop runs continuously, executing the functions in a cyclic manner

→ Foreground ⟶ ISR

- the foreground consists of ISRs that hanle async events.
- ISRs are triggered by H.W interrupts and preempt the execution of the background loop.



← Advantages:-
→ High responsiveness
→ Low power consumption
: disadvantages:.
→ low determinism → delay between ISR and background
→ to ensure timely execution of urgent actions, they must be moved to ISRs.
→ extra H.W resource → interrupt.
→ S.W Complexity

→ Real Time operating system "RTOS":- → Real Time + OS
→ real time system → execute the correct functionality at the correct time.
→ operating system →
- OS acts as the middle layer between the application software and the system hardware
- it manages the system resources and makes them available to the user applications / tasks on a need basis.

→ Real Time operating systems:
- the ability of the OS to provide a required level of service in a bounded response time.

*Types of Real-Time operating system:-

1- Hard RTOS:
- all critical tasks must be completed within the specified time duration "dead line".
- Not meeting the deadline would result in critical failures such as damage to equipment or even loss of human life.
- ex: Airbag , ABS ...
- Tolerance < 10 %

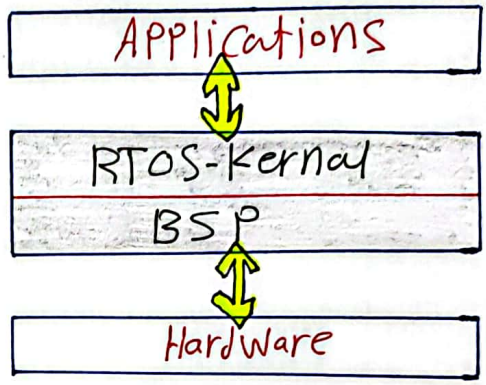
2- Soft RTOS
- less restrictive than Hard RTOS.
- if a process completes correctly, but takes longer than its given amount of time, the result may still be useful.
- ex: video games, mobile phones ...
- tolerance > 40 : 50 %

- Firm RTOS :-
  - Not meeting the deadline might not have a massive effect.
  - has a delay range between 10 : 40 %

⇒ Application of RTOS :-
- Radar gadget
- Missile guidance
- medical imaging systems
- Autopilot travel simulators. .....

structure of RTOS :-
- Kernal
- BSP

1- Kernal :



→ Kernal is the core of the operating system and is responsible for managing the system resources and the communication between the hardware and other system services.

→ the Kernal of RTOS provides an abstraction layer that hides the details of the processor hardware from the application software.

→ the Kernal contains a set of libraries and services.

→ Kernal of RTOS is referred to as a real time Kernal.

→ Kernal contains minimal set of services required for running the user application / task, such as :
- Task management
- Task scheduling
- Task synchronization
- Error / exception handling
- memory management
- Time management

2- BSP (Board support package)

→ the layer between the RTOS and the target hardware

→ it includes low-level drivers and initilization code that allows the RTOS to interface with the specific hardware platform.

→ So, it makes an RTOS target-specific.

*Difference between GPOS and RTOS

General purpose OS	Real-Time OS
→ Not deterministic: random execution pattern	→ Deterministic: No random execution pattern
→ un-predictable response time	→ predictable response time
→ Non-preemptive kernal: High priority tasks can't preempt kernal calls	→ preemptive kernal: High priority tasks can preempt low priority tasks
→ priority are dynamically adjusted	→ priority based scedwling
→ unlimited memory resources.	→ Limited memory resources.
	→ Time bound → deadline
	→ it works under worst case assumption.

* Time-Driven Vs. Event-Driven:

→ Time-Driven programming: is a computer programming paradigm نموذج, where the control flow of the computer program is driven by a clock and is often used in real time computing.

→ Event-Driven programming: is a computer programming paradigm in which the flow of the program is determined by events such as key pressed, sensor outputs or message from other programs or threads.

- Depending on the application requirements any of these paradigms is used and many times both are used in the same application

* schedulers and systick

→ scheduling is the process or algorithm which decides which task or thread should be accessed and run at what time by the system resources.

→ the system tick (systick) is the time unit that os timers and delays are based on.

- the system tick is a scheduling event. it causes the scheduler to run and may cause a context switch.

→ Task and Task Management :-

→ Real-Time embedded software applications must be designed for concurrency ~ ازاي

→ Concurrent design requires developers to divide an application into small schedulable, and sequential program units.

→ Thus, concurrent design allows system multitasking to meet performance and timing requirements for a real-time system.

→ Most RTOS Kernals provide task objects and task management services to facilitate designing concurrency within an application.

* Task :

→ A task is an independent thread of execution that can compete with other concurrent tasks for processor execution time.

→ it thinks it has the cpu all to itself

→ Developers divides applications into multiple concurrent tasks to optimize the handling of inputs and outputs within set time constraints.

→ Tasks are implemented as C-functions.

→ the task is able to compete for execution time on a system, based on a predefined scheduling algorithm.

→ upon task creation, each task has :

- A name
- A unique ID
- A priority (preemptive scheduling)
- A stack
- A task routine
- A task control block (TCB)

→ A task consists of three parts.

1- Task Program Code → ROM

2- Task stack → RAM

- residing in a RAM area that can be accessed by the stack pointer.
- the stack has the same function as in a single-task system :
  - → storage of return addresses of function calls
  - → parameters and local variables
  - → temporary storage of intermediate calculation results and register values.

3- Task Control Block (TCB):-

- A data structure assigned to a task when it is created.
- it contains information and state related to a specific task:-
  - stack pointer
  - task priority
  - task's name and ID
  - current task status.

*Task structure:-

→ any task is defined by the following parameters:-

- periodicity: how often task should be scheduled
- periority: task priority of execution in relation to other tasks in the system.
- ExecutionTime: time the task takes to run to its completion since it was scheduled
- Deadline: the time bound or limit that the task execution should not exceed.

*When writing code for tasks, tasks are structured in one of two ways:

1- run to completion 2- endless loop

→ Run to Completion:-

→ Run-to-Completion tasks are most useful for application initialization and startup

→ they typically run once, when the system first powers on.

→ it has a higher priority than the application tasks it creats so that its initialization work is not preempted.

→ the application initialization task is written so that it suspends or delets itself after it completes its work so the newly created task can run.

→ ex:

```
RunTo CompTask() {
    initialize application
    creat endless loop tasks
    creat kernal objects
    Delete or suspend this task
}
```

→ Endless loop

→ Endless-loop tasks do the majority of the work in the application by handling inputs and outputs

→ Typically, they run many times while the system is powered on.

→ the structure of an endless loop task can also contain initialization code. that only needs to be executed once when the task first runs, later than the task executes in an endless loop.

→ FreeRTOS tasks must not be allowed to return from their implementing function in any way they must not contain a return statement and must not be allowed to execute past the end of the function.

→ ex:

```
EndlessLoopTask ( ) {
    initialization code
    Loop forever {
        Body of loop
        make one or more blocking calls
    }}
```

---

* System Tasks:-

→ When the kernal first starts, it initiates a set of system tasks, each assigned a specific priority from a set of reserved priority levels, which are intended for the internal use of the RTOS.

→ An application should avoid using these priority levels for its task because running application tasks at such levels may affect the overall system performance or behavior.

→ For most RTOSes, these reserved priorities are not strictly enforced

→ examples of system tasks:

- initialization or startup task: initializes the system, creats and starts system tasks
- Idle task: consumes processor idle cycles when no other activity is present
- Logging task: logs system messages
- EXception-handling task: handles exceptions.
- debug agent task: allows debugging with a host debugger.

→ other system tasks might be created during the initialization phase, depending on what other components are included with the kernal.

* Idle task :-

→ the idle task is created at kernal startup and runs at the lowest priority.
→ it is one of the system task that bears mention and shouldn't be ignored.
→ it executes in an endless loop when no other tasks are ready to run, utilizing idle processor cycles.
→ it is necessary to keep the processor executing valid instructions even when no application tasks exist, by having the program counter point to the idle task code.
→ therefore, the idle task ensures the processor program counter is always valid when no other tasks are running.
→ in some cases, the kernal may allow running a user-configured routine instead of the idle task, for special requirements like power conservation when the system is idle.

→ there are two basic systems with different levels of support for task states.
  → the first system supports three basic states.
  → while the second system supports five states.

[1] the system with five basic states of task :-
- Dormant state
- Ready state
- Pending state
- Running state
- interrupt

[2] the system with three basic states of task :-
- Ready state
- Running state
- Blocking state

→ the Dormant state : represents tasks that are in memory but not yet available to the system
  • tasks become available by using the xTaskCreate() function.

- When it is no longer neccessary for the system to manage a task, we can call the task delete function vTaskDelete().
  - → it does not actually delete the code of the task, it is simply not eligible to access the CPU.

→ **the Ready State** : corresponds to a ready-to-run task, but is not the most important task ready.
- → there can exist multiple tasks that are ready, and the system maintains a ready list to keep track of them
- → this list, sorted by priority.

→ **The Running State** :
- → in a single CPU system, only one task can run at a time.
- → in this case, when a task is moved to the running state, the processor loads its registers with this task's context.
- → the processor can then execute the task and manipulate the associated stack
- → A task can move back to the ready state while it is running.
  - → it is preempted by a higher priority task
  - → in this case, the preempted task is put in the appropriate priority-based location in the task-ready list, and the higher priority task is moved from the ready state to the running state.

→ **The pending state** :-
- → tasks in the pending state are placed in a special list called a pend-list or wait-list, which is linked to the event that the task is waiting for.
- → when waiting for the event to occur, the task does not consume CPU time.
- → when the event occurs, the task is placed back into the ready list and the system decides whether the newly readied task is the most important ready-to-run task. if this is the case, the currently running task will be preempted (placed back in the ready list) and the newly readied task will run immediately if it is the most important task

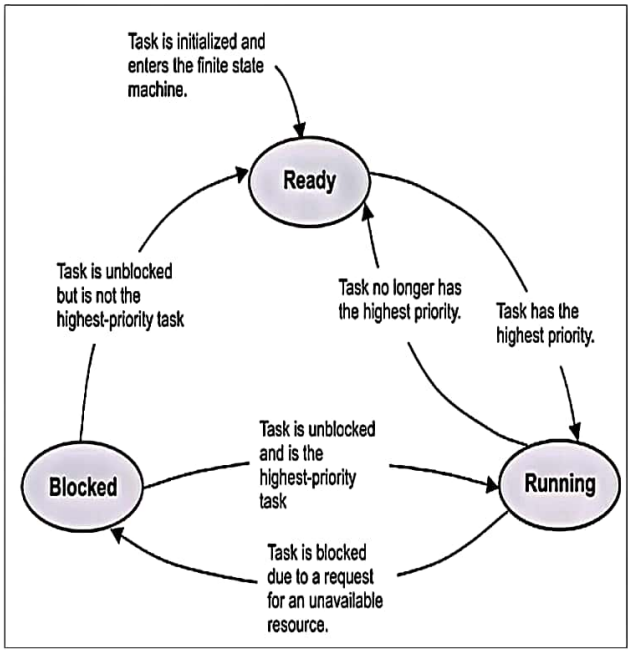
→ **Interrupt** :-

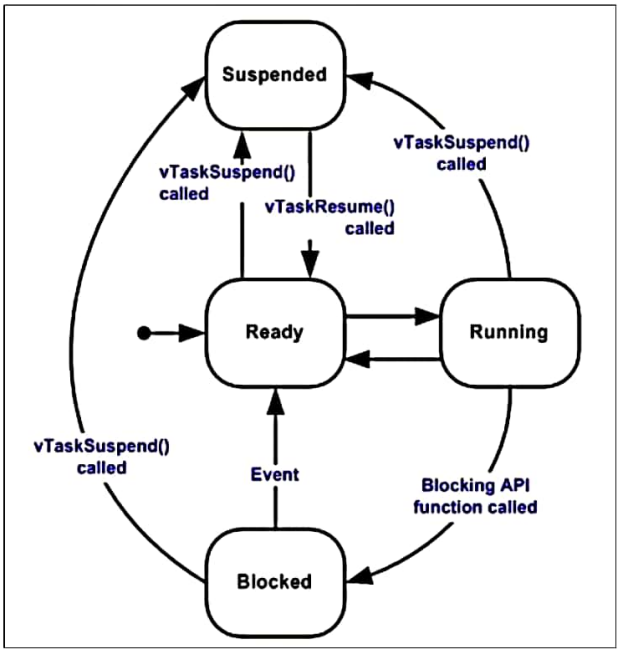
→ interrupting devices can suspend task execution and trigger an ISR
→ ISRs are typically events that tasks wait for
→ An ISR should simply notify a task that an event occurred and let the task process the event
→ ISRs should be as short as possible and most of the work of handling the interrupting devices should be done at the task level where it can be managed by RTOS.

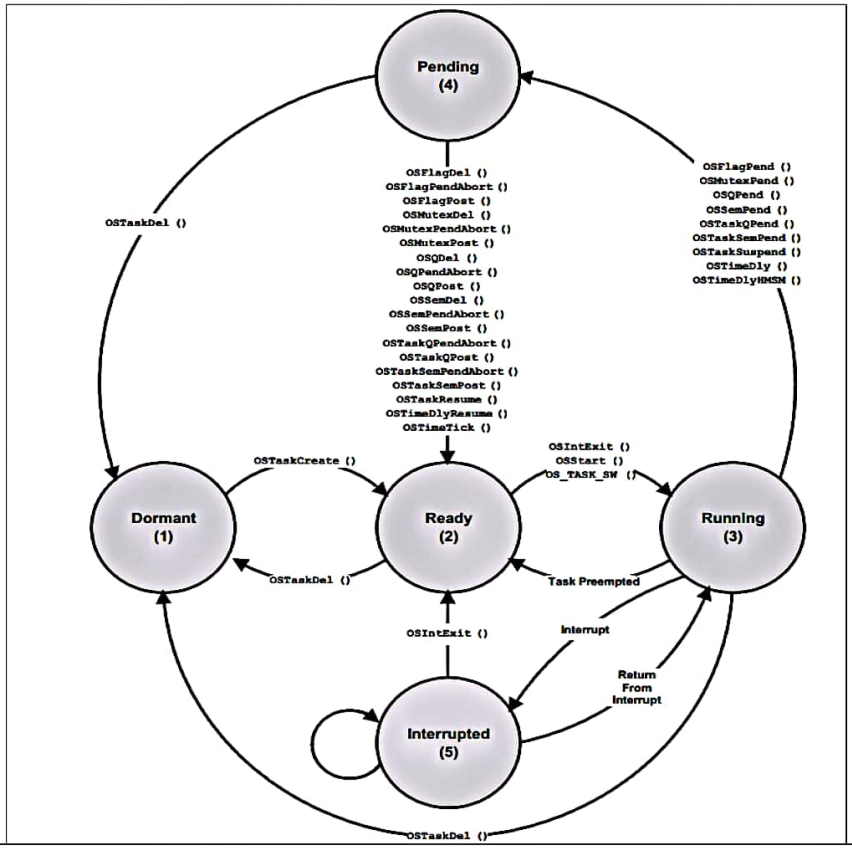
→ If we have an RTOS that supports only three states, we will come across the term "blocking state"

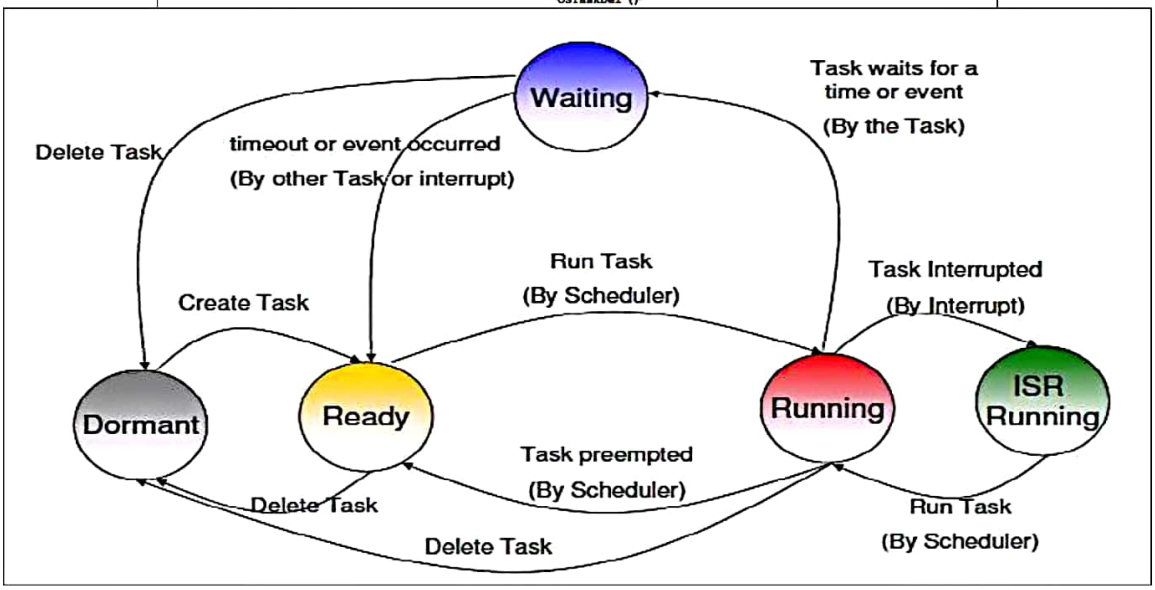
→ **Blocking state** :-

→ A task is said to be in the blocked state if it is currently waiting for either a temporal or external event
→ if a task calls vTaskDelay() it will block until the delay period has expired
→ tasks in the blocked state normally have a 'time out' period, after which the task will be timeout, and be unblocked, even if the event that the task was waiting for has not occurred.
→ tasks in the blocked state do not use any processing time and cannot be selected to enter the running state
→ suspended tasks are similar to tasks in the blocked state, but tasks in the suspended state cannot be selected to enter the running state, but tasks in the suspended state do not have a time out. instead, tasks only enter or exit the suspended state when explicitly commanded to do so through the vTaskSuspend() and xTaskResume() API calls respectively.
→ the possibility of blocked states is extremely important in RTOS.
  • without blocked state, Lower priority tasks couldn't run.
→ if higher priority tasks are not designed to block, "cpu starvation" can result
→ A task can only move to the blocked state by making a blocking call.
→ A blocked task remains blocked until the blocking condition is met.









* Scheduling and scheduling Algorithms.

→ Concept:-

→ the scheduler is at the heart of every kernal.

→ the scheduler provides algorithms needed to determine which task executes and when

→ To understand how scheduler work, we should know :-

- schedulable entities "tasks"
- Context switching
- scheduling algorithms
- Multitasking
- Dispatcher

* scheduling entities :-

→ A scheduling entity is a kernal object that can compete for CPU time based on a specific scheduling algorithm.

→ tasks and processes are all examples of schedulable entities found in most kernals.

→ tasks operate independently and can run concurrently with other tasks.

→ each task consists of a sequence of instructions that are executed by the CPU.

→ the instructions within the task are independant, allowing the task to be interrupted and resumed without impacting the system's overall operation

* Multitasking :-

→ Multitasking is the ability of the operating system to handle multiple activities within set deadlines.

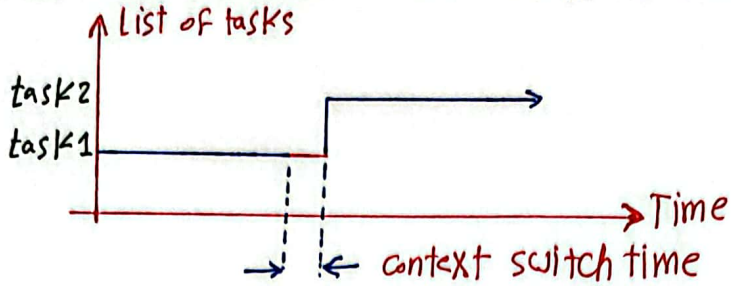
→ A real time kernal might have multiple tasks that it has to schedule to run

→ in this scenario, the kernal of the system handles multiple tasks in a way that makes it seem like they are running together at the same time. However, the kernal actually takes turns executing each task sequentially, based on a predetermined schedule.

→ the scheduler must ensure that the appropriate task runs at the right time.

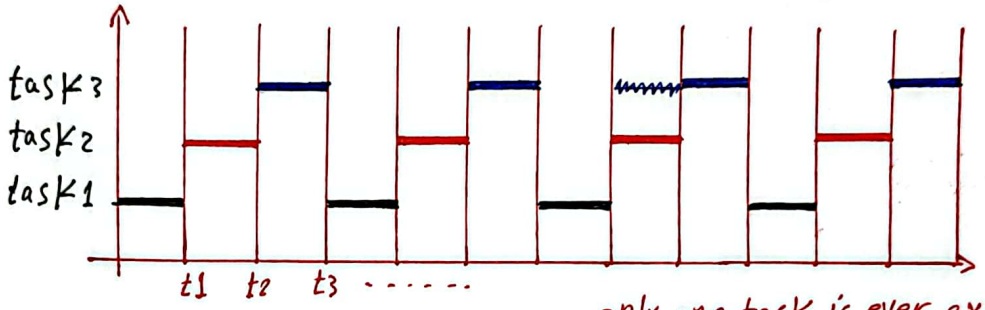
→ An important point to note is to understand that tasks operate according to the kernal's scheduling rules, while ISR are activated by hardware interrupts based on set priorities.

→ As the number of tasks increases, the CPU's workload also increases. this is because more tasks mean more frequent context switches.



*Multi tasking Vs. Concurrency :

→ A conventional processor can only execute a single task at a time, but by utilizing multitasking the OS can rapidly switch between tasks, creating the impression that multiple tasks are running concurrently.



only one task is ever executing at any time.

* Context Switching:-

→ is the process of saving the context of a task being suspended and restoring the context of a task being resumed.

→ each task has its own context, which is the state of the CPU registers required each time it is scheduled to run.

→ A context switch occurs when the scheduler switches from one task to another

→ the Kernal's job in this process is:

- every time a new task is created, the kernal also creats and maintains an associated task control block (TCB).
- TCBs are system data structures that the kernal uses to maintain task-specific-information.
- TCBs contain everything a kernal needs to know about a particular task.
- when the kernal's scheduler determines that it needs to stop running task1 and start running task2, it takes the following steps:

1- the kernal stops task1 and saves it's context information in its TCB.
2- it loads task2's context information from its TCB, which becomes the current thread of execution.
3- the context of task1 is frozen while task2 executes.

→ the time it takes for the scheduler to switch from one task to another is the context switch time
  → it is usually small compared to the time tasks spend executing their operations
  → However, when an application is poorly designed and involves excessive context switching, it can lead to performance issues due to the overhead associated with these switches
  → therefore, we should design applications in a way that does not involve excess context switching.

→ When a task is suspended or resumed just before executing an instruction, it faces uncertainties due to potential modifications made by other tasks to the register values it relies on during the suspension.
→ To address this issue, a context switch is performed by OS Kernal.
→ the context of the task including register values (SP, PC...) is saved when it get suspended and restored when it resumes. this ensure that the task has an identical context as before suspension and avoids incorrect results.

* Before discussing more about context switching, we will first remember some notes about ARM Cortex registers:-
→ R0 : R12 → general purpose registers
→ R13, stack pointer:
  • Handler Mode → MSP (main stack pointer) selected
  • thread Mode → MSP / PSP (process SP) } ← control register
→ R14, Link register (LR):
  • it is used for holding the return address when calling a function
→ R15, program counter:
  • it contains the current instruction address.

→ program status registers (PSR):
  → the PSR is composed of 3-status registers
    - Application PSR (APSR)
    - Execution PSR (EPSR)
    - interrup PSR (IPSR)
  → the combined PSR provides information about program execution and the Alu flags
  → the APSR contains the Alu flags (N, Z, C, V)
  → the IPSR contains the current executing ISR number.
  → the EPSR → T-bit → thumb state

* Context switch in details
→ Let us assume that task_B has a higher priority and is ready to execute, and the task_A has a lower priority.

* the state of the system when it is still running Task_A just before the PendSV interrupt fires.
→ all the CPU registers can have important data that shouldn't get lost
→ the arm-cortex-M has two stack pointers (PSP, MSP)
→ the Flash memory contains a pointer called "PXCurrentTCBConst" which points to the PXCurrentTCB pointer which is located in RAM - this pointer's role is tracking the currently running task.
→ TASK_A is currently running using CPU registers
→ the process stack contains information related to taskA
→ the PSP points to the top of the stack of task_A.
→ pxCurrentTCB points to TCB_A, which stores Task_A's context information

* Entering PendSV Handler:
- upon entering the PendSV_Handler() ISR, the CPU will automatically push registers (R0:R3, R12, LR, PC and PSR) onto the process stack
- After this the CPU writes an EXC_RETURN value into R14 (LR)
  → the EXC_RETURN is standardized pattern.
  → the Lowest 5-bits give informations about which stack was used.
  → at the end of ISR the EXC_RETURN value should be written into the PC.

- storing the current value of the PSP (pointing to task_A's stack) into task_A's TCB
- push the remaining core register [R4:R11, R14] onto the process stack
- then we save top of stack_A [R0_register] into the 1st member of TCB_A

→ all registers are now on the process stack, so now it is about time to start switching to task_B

→ the context switch is performed by making the global variable PxCurrentTCB points to TCB_B instead of TCB_A

→ to make this, PendSV_Handler will call another subroutine vTaskSwitch Context().

→ vTaskSwitch(): → making PxCurrentTCB points to TCB_B

* before returning from pendsv_Handler:-

→ all core register popped from task_B stack, restoring the state of taskB

→ retur from interrupt

* running taskB:

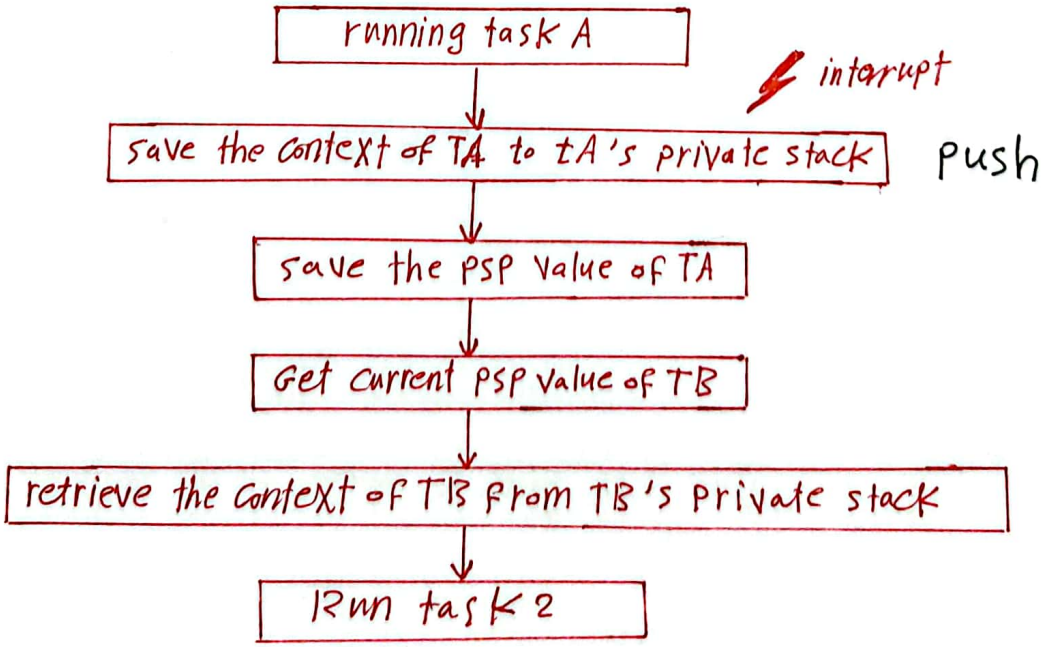
→ Cpu registers → Hold the state of task_B

→ Process stack → Contains taskB's stack Prome and data

→ Px currentTCB → Points to the TCB_B

→ TCB_B → Contains task_B's context information

* summarization:-



* Scheduling Algorithms :-

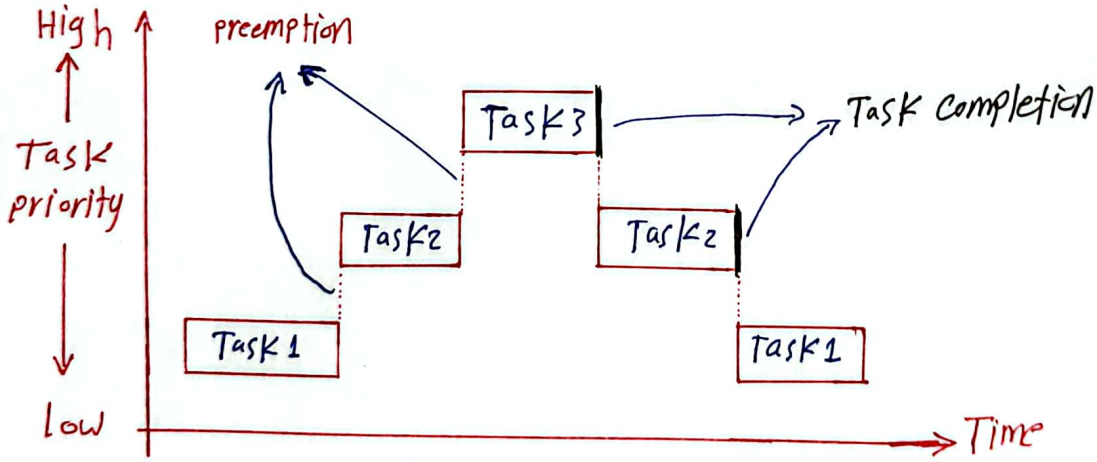
→ the scheduler determines which task runs by following a schedule algorithm also known as scheduling policy.

* Types of Kernals :

1- preemptive scheduling 2- Non-preemptive scheduling (cooperative)

1- preemptive

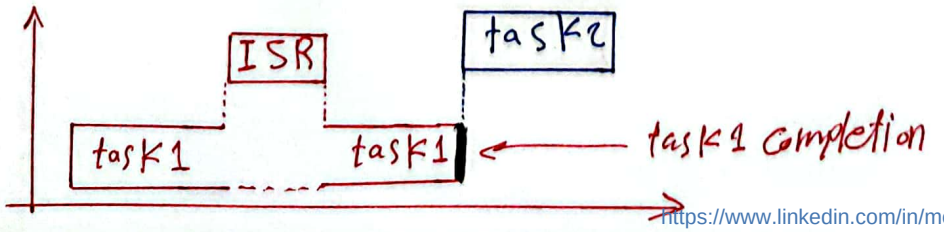
- tasks are mostly assigned with their priorities
- RTOS runs a task with a higher priority before another lower priority task, even if the lower priority task is still running, it will hold for some time and resume when the higher priority task finishes its execution.
- most real-time kernals use preemptive priority-based scheduling by default.
- real-time kernals generally support 256 priority levels, in which 0 is the highest and 255 the lowest.
- If a task with a priority higher than the current task becomes ready-to-run, the kernal immediately saves the current task's context in its TCB and switches to the higher-priority task.



2- Non-preemptive scheduling :: (Co-operative schedule) :-

→ the OS never interrupts a running process to initiate a context switch from one task to another

→ processes must voluntarily yield control periodically or when logically blocked on a resource.



non-preemptive scheduling	preemptive scheduling
• an ISR can make a higher priority task ready-to-run, but the ISR always returns to the interrupted task	• the ready-to-run higher priority task gains control of the cpu once the ISR complete it's execution
• the new high priority task gains control of the cpu only when the current task gives up the cpu	• the higher priority task preempts the lower priority task.
• low system responsiveness	• High system responsiveness
• Non-Reentrant functions can't be use	• Non-Reentrant functions can be used
• less need to guard the shared resource	• High need to guard the shared resource
• not recommended for rtos	• recommended for rtos
• interrupt latency is low	• Starvation might occur.

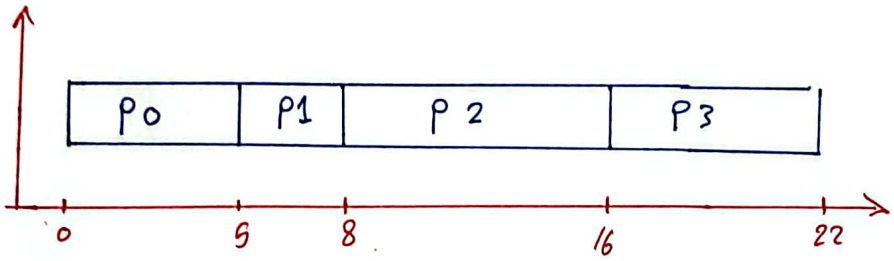
* Atomic operations :
  → operations that can not be interrupted
  → not all cpu architectures have native support for atomic operations.
  → on the kernal side atomic operations are very easy to emulate :-
    1- Disable interrupts temporarily
    2- Do your operation
    3- enable interrupts again.

* Some scheduling algorithms :-
  - First come first serve
  - Shortest job first
  - Highest priority first
  - Shortest remaining time (earliest deadline first)
  - Round Robin
  - Rate Monotonic

1- First come first serve :-

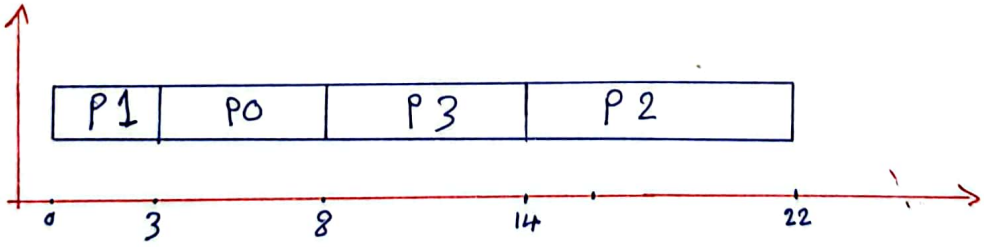
→ tasks are executed on first come, first serve basis.
→ it is (non-preemptive, preemptive) scheduling algorithm.
→ its implementation is based on FIFO queue.
→ Poor in performance as average wait time is high

Tasks	Arrival time	Execute time
P0	0	5
P1	1	3
P2	2	8
P3	3	6



2- Shortest Job first :-

→ this is a (non-preemptive, preemptive) scheduling algorithm.
→ Best approach to minimize waiting time.
→ the processor should know in advance how much time process will take



3- Highest priority first :-

→ one of the most popular scheduling algorithms.
→ each task is assigned a priority level
→ the task with the highest priority will be given the opportunity to use the CPU
→ in the preemptive version of the algorithm, a running task can be interrupted if a higher priority task enters ready-to-run state
→ in the non-preemptive one, once a task is started it can't be interrupted by a higher priority task.

→ tasks with same priority are executed on first come first served or round-robin scheduling.

4- shortest remaining time (earliest deadline first) :-

→ it is a dynamic-priority driven algorithm, in which highest priority is assigned to the request that has the earliest deadline, and a higher priority task always preempts a lower-priority one.

5- Round-Robin :-

→ it is a preemptive type of scheduling algorithm
→ there are no priority assigned to tasks.
→ each task is put into a running state for a fixed predefined time. this time is commonly referred to as time-slice (aka quantum)
→ a task can not run longer than the time slice
→ in case a task has not completed by the end of its dedicated time-slice, it is interrupted, so the next task from the scheduling queue can be run in the following time slice
→ A preempted task has an opportunity to complete its operation once it's again its turn to use a time-slice
→ No starvation, no guarantees to meet deadlines

FreeRTOS API conventions and CMSIS_OS API

* Prefixes at Variable names :

- C → char
- S → short
- L → Long
- X → portBase_TYPE (user-defined) defined in portmacro.h
  - in STM32 → Long
- U → unsigned
- P → pointer

* function name structure:-

prefix	+	filename	+	function name
Void ← V	+	task	+	task create
return port_Base ← X		queue		taskDelete
private ← prv		List		⋮
		timers		
		⋮		

ex:

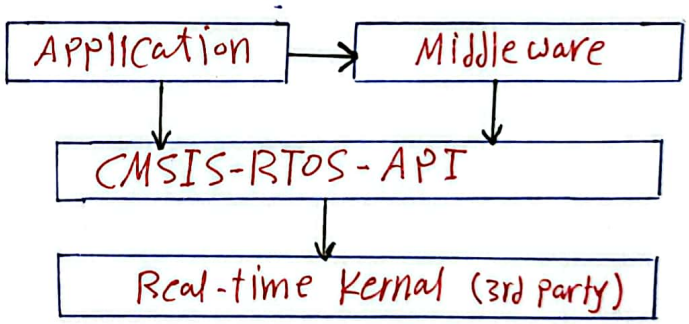
Void vTaskPrioritySet (TaskHandle_t xTask, uBaseType_t uxNewPriority

→ ux → unsigned + PortBase_TYPE
→ vTask... → Void ↔ task.h ↔ Functionality

→ typedef long BaseType_t;
→ typedef unsigned long uBaseType_t;

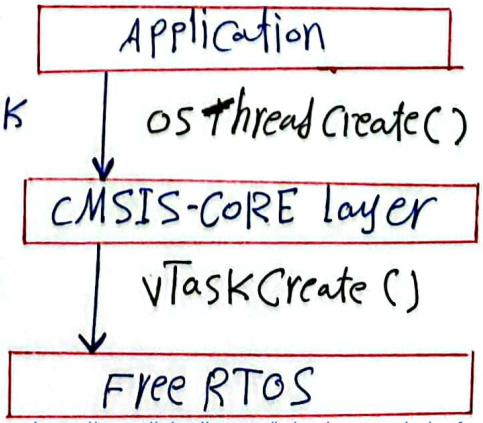
* Prefixes at macros defines their definition location:
→ Port → ex. portMAX_DELAY → portable.h
→ task → ex. task_ENTER_CRITICAL → task.h
→ pd → ex. pdTRUE → projdefs.h
→ config → ex. configUSE_PREEMPTION → FREERTOSConfig.h
→ err → ex. errQUEUE_Full → projdefs.h

* CMSIS_OS API:-
→ the main point is that the CMSIS-OS API acts as a "simplified additional Layer" on top of the FreeRTOS Kernal's API.



→ this means developers can use the CMSIS-OS API to interact with FreeRTOS without needing to know all the intricate (complex) details of the FreeRTOS API itself.
→ this abstraction makes it easier to port applications across different RTOS platforms, as the CMSIS-OS API provides a standardized interface.

ex:
→ osThreadCreate(); an API provided by CMSIS-RTOS layer to create an RTOS Task (independant of an underlying RTOS)
→ vTaskCreate(): the actual API provided by FreeRTOS to create a task (specific)



FreeRTOS Memory Management

* Memory Allocation schemes :-

→ FreeRTOS provides the option to choose between static and dynamic memory allocation for creating RTOS objects.

1- Static allocation :-

→ Memory is allocated during compile-time, this provides more control to the developer.

→ RTOS object can be placed at specific memory locations.

→ the maximum RAM footprint can be determined at Link time, rather than run time

→ there is no need to handle memory allocation failures

→ suitable for environments with restrictions on dynamic memory allocation.

2- Dynamic allocation :-

→ Memory is allocated during runtime, this method provides simplicity and the potential to optimize the application's RAM usage.

→ simplicity : fewer function parameters are required when creating an object dynamically.

→ the memory allocation occurs automatically, within the RTOS API functions.

→ So, the application writer does not need to concern themselves with allocating memory themselves.

→ When an RTOS object is deleted, the memory used by that object can be reused, potentially reducing the application's maximum RAM footprint.

→ RTOS API functions provide information on heap usage, enabling optimization of the heap size.

→ the application can choose a memory allocation scheme that best suits its requirements, such as heap_1 for simplicity and determinism often necessary for safty critical applications, heap_4 for fragmentation protection, heap_5 to split the heap across multiple RAM regions, or an application scheme provided by the application writer themselves.

* Both methods have pros and cons, and both methods can be used within the same RTOS application, allowing for a customized memory management approach.

* there are five different heap implementations, named "heap 1-5" available in FreeRTOS.

→ to choose which heap allocation to use, change the USE_HEAP define in the heap.C file

* Heap_1.C implementation :-

→ Heap_1 is similar to static allocation in the way that once the memory is taken, it cannot be freed or reallocated.

→ it is less useful since FreeRTOS added support for static allocation.

→ this implementation subdivides a single array into smaller blocks as RAM is requested.

④ → the total size of the array (total size of the heap), is defined by the configTOTAL_HEAP_SIZE define, which is defined in FreeRTOSConfig.h

⑤ → the configAPPLICATION_ALLOCATED_HEAP configuration constant allows the heap to be placed at a specific memory address.

⑥ → the xPortGetFreeHeapSize() function provides information about the remaining unallocated heap size.

→ Heap_1 suitable usage scenarios:

- if the application never deletes a task, queue.....
- the implementation always takes the same amount of time to execute "Deterministic" and avoids memory fragmentation
- it can be used in application that do not support true dynamic memory allocation.

---

* Heap_2.C implementation :.

→ heap_2 uses a best fit algorithm and allows previously allocated blocks to be freed. it does not combine adjacent free blocks into a single large block.

→ points ④, ⑤, ⑥ of Heap_1 are the same in Heap_2, but Heap_2 does not provide information on how the unallocated memory is fragmented into smaller blocks.

→ pvPortCalloc() function : this function allocates memory for an array of objects and initializes all bytes in the allocated storage to zero. it has the same signature as the standard library calloc function

* Heap_2 usage considerations :-

→ Heap_2 allows for the deallocation of previously allocated blocks, making it suitable for applications that repeatedly create and delete tasks, queues--etc

→ it may lead to memory fragmentation and allocation failures when dealing with variable-sized memory

→ the application can directly call pvPortMalloC() and vPortFree() instead of using them indirectly through other FreeRTOS API functions.

→ heap_2 is not deterministic, but it is more efficient than most standard C library malloc implementation

→ suitable for small real-time systems that require dynamic object creation

→ not recommended for new projects and kept due to backward compatibility.

---

* Heap_3 implementation :

→ the heap for FreeRTOS is located in the free RAM area outside the stack and heap areas for the main application

→ the only exception is the heap_3 model, which is located in the heap area of main application

→ in heap_3 model we have full control on the memory allocation by creating our own linker script file and our own methods to allocate and deallocate the memory.

→ the heap_3 uses the malloc() and free() functions, so the size of the heap is defined by the linker configuration, and the configTOTAL_HEAP_SIZE has no effect when heap_3 is used.

→ heap_3 will probably considerably increase the RTOS kernal code size.

→ it is not recommended for both systems, the critical and limited resources systems.

→ requires the linker to setup a heap, and the compiler library to provide malloc and free() implementations.

→ it is not deterministic.

---

* Heap_4 implementation :- (not deterministic - but is much more efficient)

→ the most popular one.

→ it uses first fit algorithm to allocate memory

→ it is able to combine adjacent free blocks into a single larger block, which minimize the risk of memory fragmentation.

→ the memory array for the heap is declared within the heap_4.c file itself.
→ the starting address of the heap memory array is automatically configured by the linker
→ if configAPPLICATION_ALLOCATED_HEAP is set to zero, a static declaration of the heap array occurs within heap_4.c, and its size is determined by configTOTAL_HEAP_SIZE
→ FreeRTOS allows for custom heap allocation by the application writer
→ To use a custom memory area for the FreeRTOS heap:-
- Set configAPPLICATION_ALLOCATED_HEAP to 1 in the FreeRTOSConfig.h file
- Declare the heap memory array within the user's code, specifying the desired start address and size using configTOTAL_HEAP_SIZE

→ heap_4 : heap memory is organized as a Linked List → for better efficiency when dynamically allocating / freeing memory.
→ When allocating "N" bytes in the heap memory using "pvPortMalloc" API, it concumes:
→ 8 bytes for the structure of the "Linkedlist heap block"
→ N bytes for the data that needs to be allocated
→ Padding for 8 bytes alignment.
* ex: if we need to allocate 52 bytes
→ 8 + 52 = 60 bytes → aligned to 8 bytes → it gives 64 bytes consumed from the heap.

---

* Heap_5 implementation :-
→ the algorithm used by heap_5 to allocate and free memory is identical to that used by heap_4
→ unlike heap_4, heap_5 is not limited to allocating memory from a single statically declared array, it can allocate memory from multiple and separated memory spaces.
→ heap_5 is used in case we have MCU with separated RAM areas that not visible as continuous memory space.
→ it is able to combine adjacent free memory blocks into a single block like Heap_4

→ it is the only memory allocation scheme that must be explicitly initialized before any OS object can be created → before first call of pvPortMalloC().
→ to initialize this scheme vPortDefineHeapRegions() function should be called
- it specifies start address and size of each separate memory area that we would like to dedicate as a heap for our os application.

---

* FreeRTOS memory allocation depends on stack operation models:
→ Cortex-M processor uses the "Full Descending stack" model
→ each push operation, the processor first decrements the SP, then stores the value in the memory location pointed by SP
→ the SP points to the memory location where the last data was pushed to the stack
→ each Pop operation, the value of the memory location pointed by SP is read, and then the value of SP is incremented automatically.
→ if the stack grows down then allocate the stack then the TCB so the stack does not grow into the TCB. likewise if the stack grows up then allocate the TCB then the stack

---

* Creating task in details:.
→ To add a new task to the list of tasks ready-to-run:
- make sure that the configuration option configSUPPORT_DYNAMIC_ALLOCATION in the file FreeRTOSConfig.h is set to "1". if it is left undefined, it will default to "1" which is also acceptable.
- each task requires a certain amount of RAM to hold its state and serve as its stack. the allocation of this RAM depends on how the task is created:
  → if the task is created using the function xTaskCreate(), the necessary RAM is automatically allocated from the FreeRTOS heap.
  → if the task is created using the function xTaskCreateStatic(), the application writer needs to provide the RAM, allowing for static allocation at compile time

→ Task creation means creating a task in the memory (allocating memory for TCB)
→ Task implementation is the one which runs on the CPU.

* We first create a task
  → this is done by using xTaskCreate(), which is an API provided by freeRTOS to create a task
  → this API creats a new FreeRTOS task using memory allocation and adds the newly created task (TCB) to ready queue of the kernal.

```
BaseType_t xTaskCreate ( TaskFunction_t pvTaskCode,
                         const char * const pcName,
                         const configSTACK_DEPTH_TYPE uxStackDepth,
void * const pvParameters ,   UBaseType_t uxPriority,
                         TaskHandle_t * const pxCreatedTask )
```

* Parameters :
  → pvTaskCode :
    → This is a pointer to the task entry function, which is simply the name of the function that implements the task.
    → Tasks are typically implemented as infinite loops, and the function must not attempt to return or exit. However, tasks have the ability to delete themselves
  → pcName :
    → A descriptive name for the task
    → This is mainly used to facilitate debugging.
  → uxStackDepth :
    → Specify the number of words (not bytes) to allocate for the task's stack.
  → pvParameters :
    → pointer of the data which needs to be passed to the task handler once it gets scheduled.
    → if pvParameters is set to the address of a variable, then the variable must still exist when the created task executes. So it is not valid to pass the address of a stack variable.
  → uxPriority :
    → Specify the priority at which the task will execute
    → Lower the priority value, lesser the task priority
    → in FreeRTOS each task can be assigned a priority value from "0" to (configMAX_Priorities -1) where configMAX_PRIORITIES is defined in FreeRTOSConfig.h

→ you must decide configMAX_PRIORITIES as per your application need.
→ using too many task priority values could lead to RAM's overconsumption, and decrease the system's overall performance.

→ PxCreatedTask:
→ this parameter is optional when creating a task
→ By passing the address of a variable of type TaskHandle_t as the value of PxCreatedTask, we can obtain a handle to the created task.
→ this handle can be used to perform various operations on the task, such as suspending, resuming, or deleting it
→ this parameter can set to Null.

* after finishing the creation of the task, we will implement it.

→ Task implementation:
→ the task implementation also called task handler or task function.
→ task implementation is the one which runs on the CPU.

```
void vFirstTask ( void * pvParameters) {
    for( ; ;) {
        /* task application code */
    }
    vTaskDelete (NULL);
}
```

→ A task will normally be implemented as an infinite loop
→ if the task doesn't implement infinite loop or in any case if it breaks the loop to go out of the task function, then the task must be deleted before exiting the task function.
→ So, task handler should never return without deleting the associated task.

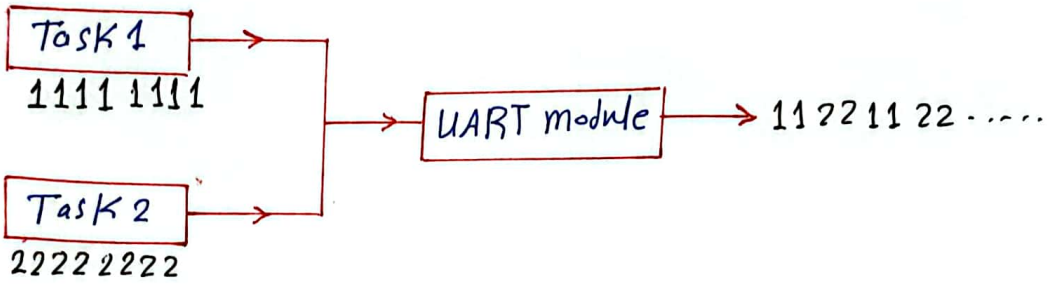
xTaskCreate()

TCB	Task's stack	Ready list
→ TCB will be created in RAM (Heap section) and initialized.	Dedicated stack memory will be created for a task and initialized. This stack memory will be tracked using PSP register of the processor	task will be put under ready-to-run list for scheduler to pick.

Synchronization of tasks
(inter task access synchronization)

* Task synchronization: making processes aware of the access of shared resources by each process to avoid conflicts and unexpected results.
  → an application's design typically involves multiple concurrent threads or tasks
  → Coordinating these activities requires inter-task synchronization, this is because each task is executing independently.

* example that shows a scenario where two tasks try to access a shared resource (the UART module), without inter-task synchronization
  → this can lead to race condition and unpredictable behavior.



* another example: when two tasks with different priorities accessing the same global variable.
  → in this situation, the function must be reentrant.

→ A shared resource: is a resource which is accessed by one or more tasks.
→ it can be data, e.g. a common buffer between two tasks (global variable)
→ it also can be a hardware, e.g. LCD, UART, or an ADC..

* Critical section: which is a sequence in the program code of a task where a shared resource is used
  → relatively indivisible critical section: which is a critical section where the task is allowed to be interrupted but no other task is allowed to use the shared resource.
  → absolutely indivisible critical section: which is a critical region where the task is not allowed to be interrupted.

* Race condition:
  → A race condition is a situation that may occur inside a critical section
  → This happens when the result of multiple thread execution in a critical section differs according to the order in which the threads execute.
  → it occurs when multiple threads read/write the same variable, e.g. they have access to some shared data and they try to change it at the same time. in such a scenario threads are "racing" each other to access/change the data.
→ Race condition in critical section can be avoided if the critical section is treated as an atomic instruction.

* Thread-Safe:
  → is the term we use to describe a program, code, or data structure free of race conditions when accessed by multiple threads.
  → in order to prevent race condition from occurring, we'd typically put a Lock around the shared data to ensure only one thread can access the data at a time.

→ synchronization is classified into two categories:
  1- resource synchronization: determines whether access to a shared resource is safe, and if not, when it will be safe.
  2- Activity synchronization: determines wether the execution of a multi-threaded program has reached a certain state, and if it hasn't, how to wait for and be notified when this state is reached.

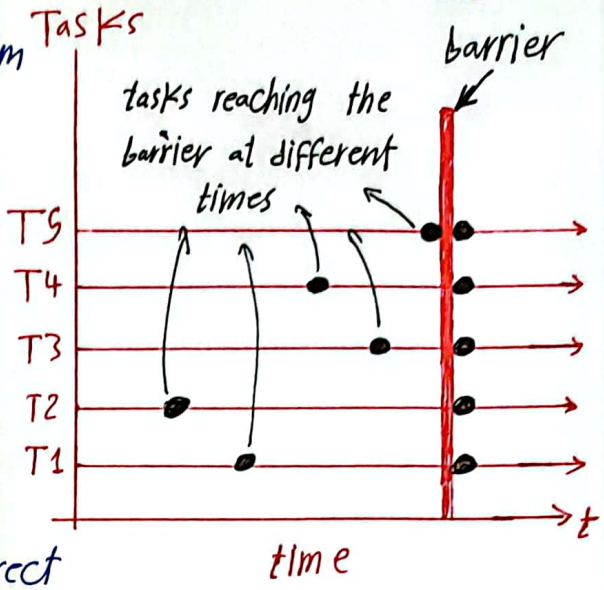
→ resource synchronization → this is what we previously explained.

* Activity synchronization:
  → task must synchronize its activity with other tasks to execute a multi-threaded program properly.
  → Activity synchronization is also called condition synchronization or sequence control
  → Activity synchronization ensures that the correct execution order among cooperating tasks is used.
  → one representative of activity synchronization methods is barrier synchronization.

→ in barrier synchronization, tasks in a system collaborate on a complex computation divided among them.

→ different tasks handle various aspects of the computation

→ these tasks must synchronize at a barrier point to combine their partial results for the final calculation. even if a task finishes its part early, it must wait for all other tasks to complete before proceeding to ensure the correct sequence of computations.



→ Mutual exclusion:

→ is a provision by which only one task at a time can access a shared resource

→ it ensures that one task's execution of a critical section is not interrupted by the competing critical sections of other concurrently executing tasks.

* Resource synchronization Methods:

1- Interrupt Locks:

→ interrupt locking (disabling system interrupts) is the method used to synchronize exclusive access to shared resources between tasks and ISRs

→ some processors allow for a fine-grained interrupt lock, where only interrupts at or below a certain level are blocked.

→ other processors only allow a coarse-grained lock, where all system interrupts are disabled

→ When interrupts are disabled, even the operating system's scheduler can not run, as the system becomes unresponsive to external events that could trigger task re-scheduling.

→ this ensures the current task continues to run until it voluntarily gives up control

→ interrupt locks are the most powerful synchronization method, but using them excessively can introduce unpredictability into the system.

→ therefore, interrupt locks should be kept short and only used when necessary to protect a task's critical region from interrupt activities.

→ A task that enables interrupt locking must avoid blocking operations, as the behavior when blocking depends on the RTOS implementation.
→ some RTOSes will block the task and then re-enable interrupts, disabling them again when the task is ready to run.
→ RTOSes that don't support this feature can cause the system to hang indefinitely.

2- preemption locks:
→ preemption locking (disabling the kernal scheduler) is another method used in resource synchronization.
→ A task disables the kernal preemption when it enters its critical section and re-enables the preemption when finished.
→ the executing task cannot be preempted while the preemption lock is in effect.
→ on the surface (للوهلة الأولى), preemption locking appears to be more acceptable than interrupt locking. but closer examination reveals that preemption locking introduces the possibility for priority inversion.
→ Although interrupts remain enabled during preemption locking, event servicing is usually delayed to a dedicated task outside the context of the ISR
→ the ISR must notify that task that such an event has occurred.
→ in most RTOS environments, when a task makes a blocking call while preemption is disabled, another task is scheduled to run, and the scheduler disables preemption after the original task is ready to resume execution.

→ RTOS uses the following methods to achieve the synchronization:-
- inter task access synchronization:
  - Semaphore
    1- Binary semaphore
    2- Counting semaphore
  - Mutex
- inter task event synchronization:
  - event flags

---

* Binary semaphore :

→ Multiple concurrent tasks of execution within an application must be able to synchronize their execution and coordinate mutually exclusive access to shared resources.

→ to address these requirements, RTOS kernals provide a semaphore object and associated semaphore management services.

→ A semaphore is a kernal object or you can say kernal service, that one or more threads of execution can acquire or release for the purpose of synchronization or mutual exclusion.

→ When a semaphore is first created, the kernal assigns to it an associated semaphore control block (SCB), a unique ID, a value (binary or a count), and a task-waiting list.

→ the value of a binary semaphore can be either 1 or 0

→ as the name indicates, this semaphore only works on 2 values (1 or 0).

→ the value of semaphore determines how many semaphore tokens are available.

- → if value = 4, then 4 semaphore keys or tokens are currently available
- → if a task tries to acquire the key for the 5th time, then it will be blocked
- → this value decreases when the key is acquire and it increases when the semaphore token or key is given back.

→ in binary semaphore, if the value = 1, then the semaphore key is available, and if the value = 0, then the key is not available.

→ if the value = 0, then the semaphore has no tokens left. at this time, if any task tries to take the semaphore, then it will be blocked.

→ Blocked tasks are kept in the task-waiting list in order either FIFO or highest priority first order.

→ when an unavailable semaphore becomes available, the kernal allows the first task in the task-waiting list to acquire it.

→ Binary semaphores are treated as global resources, which means they are shared among all tasks that need them.

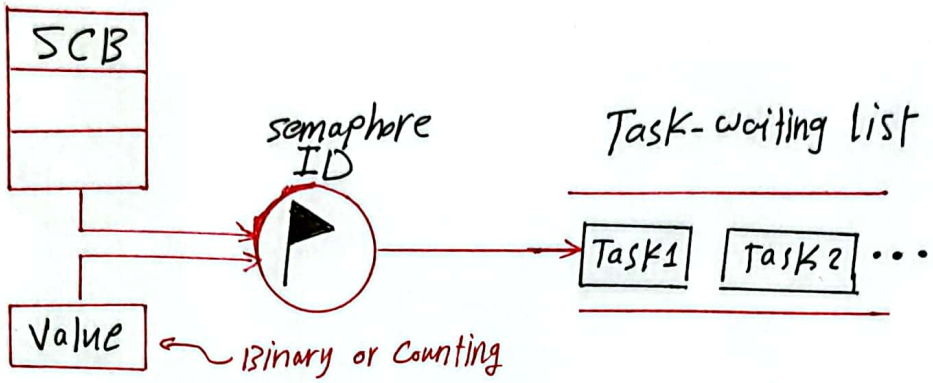
→ Making the semaphore a global resource allows any task to release it, even if the task did not initially acquire it.

→ the main use of binary or counting semaphore is synchronization, which can be between tasks or between a task and an interrupt. it can also be used for mutual exclusion, that is to guard the critical section.

* Counting semaphore:

→ it is a type of semaphore that allows more than two tasks to access the shared resource at the same time

→ Counting semaphore's values range from [0:n], where n refers to a non-negative integer.



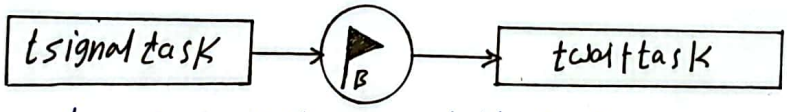
Semaphore use Cases

1- wait and signal synchronization:

→ A binary semaphore is used to coordinate the transfer of execution control between 2 tasks, tSignalTask and tWaitTask.

→ two tasks can communicate for the purpose of synchronization without exchanging data

Binary semaphore



→ the binary semaphore is initially unavailable (value=0).

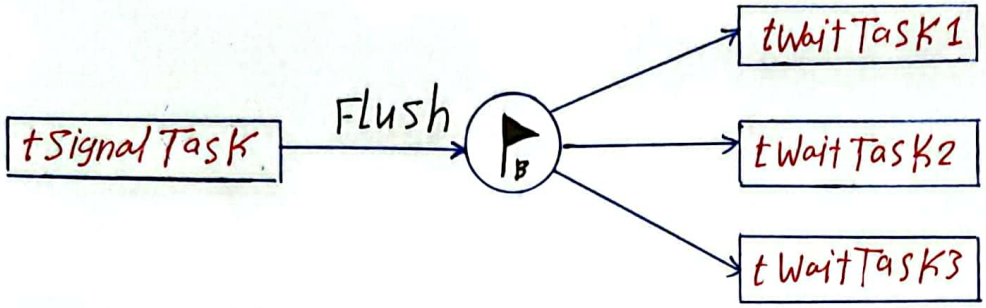
→ tWaitTask:

- has a higher priority and runs first
- makes a request to acquire the semaphore but is blocked because the semaphore is unavailable.

→ execution flow:.

- this blocking allows the lower priority tSignalTask to run.
- at some point, tSignalTask will release the binary semaphore, unblocking tWaitTask
- Because tWaitTask has a higher priority, it preempts tSignalTask and starts to execute as soon as the semaphore is released.

2- Multiple tasks waiting on signal synchronization



→ the binary semaphore is initially unavailable (value = 0)

→ Multiple tWaitTask (1, 2 and 3) have higher priorities and do some processing, when they try to acquire the unavailable semaphore, they block.

→ this blocking allows tSignalTask to complete its processing

→ tSignalTask then executes a flush command on the semaphore, effectively unblocking all three tWaitTasks

→ When coordinating the synchronization of more than two tasks, use the Flush operation on the task-waiting list of a binary semaphore.

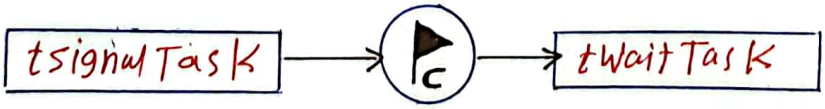
→ as soon as the semaphore is released, one of the higher priority tWait tasks preempts tsignaltask and starts to execute.

→ the value of the binary semaphore after the flush operation is implementation-dependant

→ the return value of the acquire operation must be checked to see if it is a return-from-flush or an error condition.

3- Credit tracking synchronization using a counting semaphore :

→ the counting semaphore is used when the rate at which the signaling task execution is higher than the signaled task execution.



→ the tWaitTask has a lower priority than tSignalTask

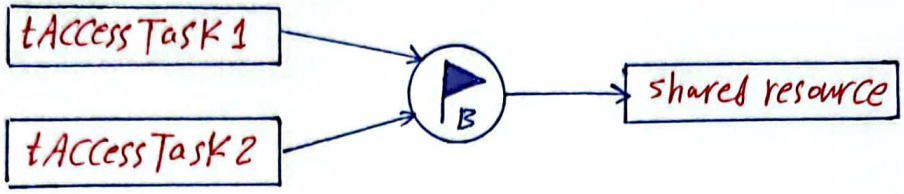
→ the counting semaphore's count is initially 0, making it unavailable.

→ the lower priority tWaitTask tries to acquire the semaphore but blocks until tSignalTask makes the semaphore available by performing a release on it

→ even then, tWaitTask will wait in the ready state until the higher priority tSignalTask eventually relinquishes the CPU by making a blocking call or delaying itself.

4- single shared resource access synchronization

→ one of the more common uses of semaphores is to provide mutually exclusive access to a shared resource.



→ a shared resource might be a memory location, a data structure, or an I/O device.

→ a semaphore can be used to serialize access to a shared resource.

→ a binary semaphore is initially created in the available state (value=1) and is used to protect the shared resource.

→ To access the shared resource, task1 or 2 needs to acquire the binary semaphore before reading or writing to the shared resource

→ if task1 executes first, it makes a request to acquire the semaphore and is successful because the semaphore is available.

→ the task1 after acquired the semaphore, it is granted access to the shared resource and can read/write to it.

→ task2 has a higher priority than task1

→ task2 tries to access the same semaphore but is blocked because task1 currently has access to it

→ after task1 releases the semaphore, task2 is unblocked and starts to execute

→ Potential issue: any task can accidentally release the binary semaphore, even if it never acquired it in the first place.

→ if this issue were to happen in this scenario, both task1 and task2 could end up acquiring the semaphore and reading/writing to the shared resource at the same time, leading to incorrect program behavior.

→ Solution: use a mutex semaphore instead.

→ Because a mutex support the concept of ownership.

→ it ensures that only the task that successfully acquired (locked) the mutex can release (unlock) it.

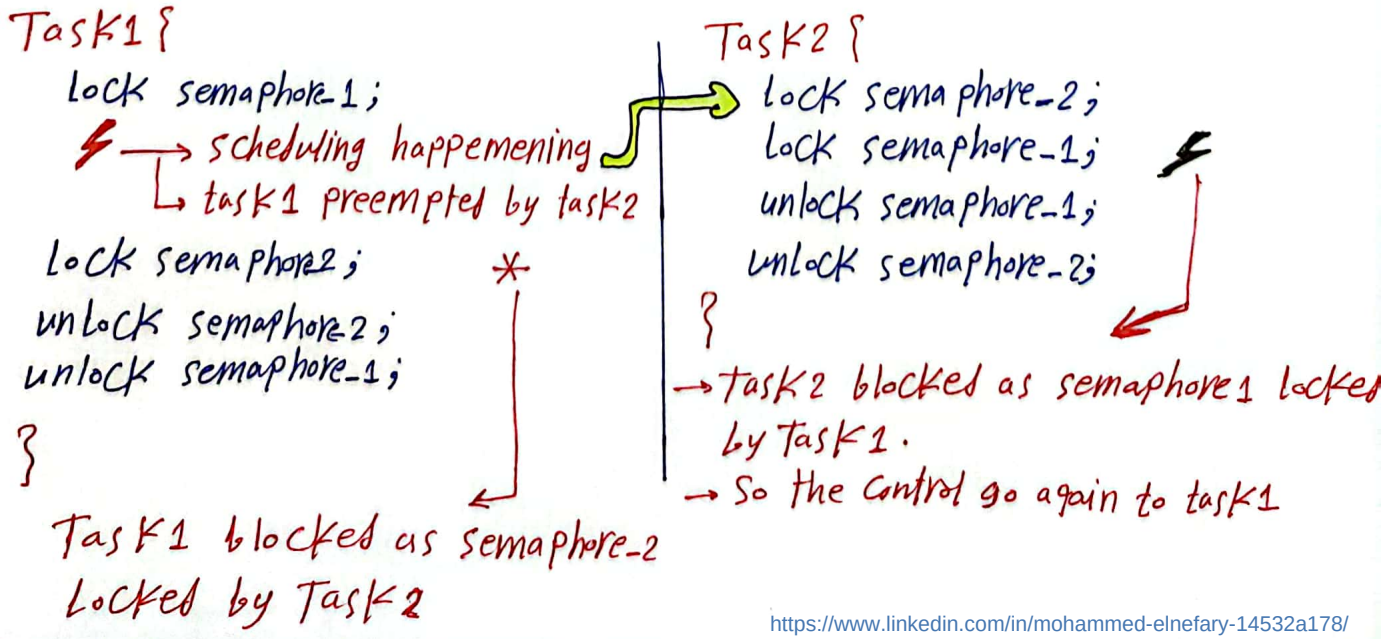
---

* Semaphore problems :
  - Starvation - Deadlock - priority inversion

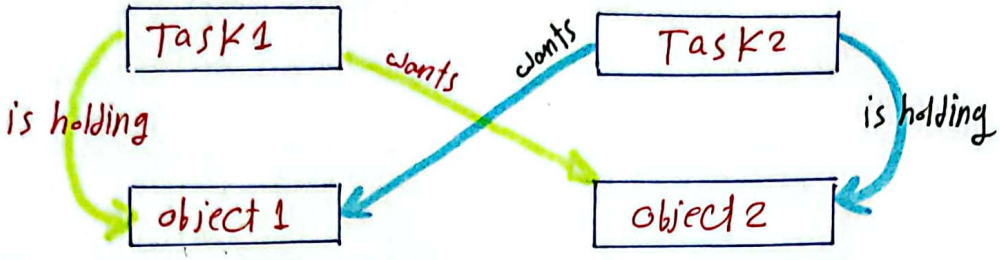
* Starvation :-
→ A low priority task will never have the chance to acquire / release the semaphore due to its priority and the scheduling decisions.
→ there are a few ways to combat starvation :
1- make sure that the high priority tasks always yield some time to the processor (e.g. by waiting for a semaphore/mutex or through a delay function to put itself to sleep)
2- the scheduler or the higher priority task monitor how long other tasks have been asleep.
→ if a lower-priority task has been asleep (blocked) for some time, its priority level is gradually raised until it gets a chance to run.
→ once it has run for some time, its priority level is dropped back to its original level.
→ this is known as "aging".

---

* Deadlock :-
→ Tasks waiting for each other to release the semaphores.
→ Deadlock is the situation in which multiple concurrent threads of execution in a system are blocked permanently because of resource requirements that can never be satisfied.
→ ex:



→ at the end of this scenario, task1 is waiting for a semaphore held by task2, and task2 is waiting for a semaphore held by task1. so Deadlock has occurred because neither task can proceed.



→ Deadlock occurs when the following conditions are present :

1- Mutual exclusion :
- A resource can be accessed by only one task at a time

2- Hold and wait :
- a task holds already-acquired resources, while waiting for additional resources to become available

3- circular wait :
- a circular chain of two or more tasks exists, in which each task holds one or more resources being requested by a task next in the chain

→ to avoid Deadlock :-

1- acquire all resources before proceeding

2- acquire the resources in the same order, and release the resources in the reverse order

→ Most kernals allow you to specify a timeout when acquiring a semaphore. this feature allows a deadlock to be broken.

---

*Task priorities :-

→ each task is assigned a priority.

→ the higher the priority given the task, the more important is that task.

→ there are 2 types of assigning task priorities :

- static priorities
- Dynamic priorities

*static priorities :-

→ Task priority is static when the task is assigned a priority that doesnot change during runtime

*Dynamic priorities :- when a task can change its priority during runtime

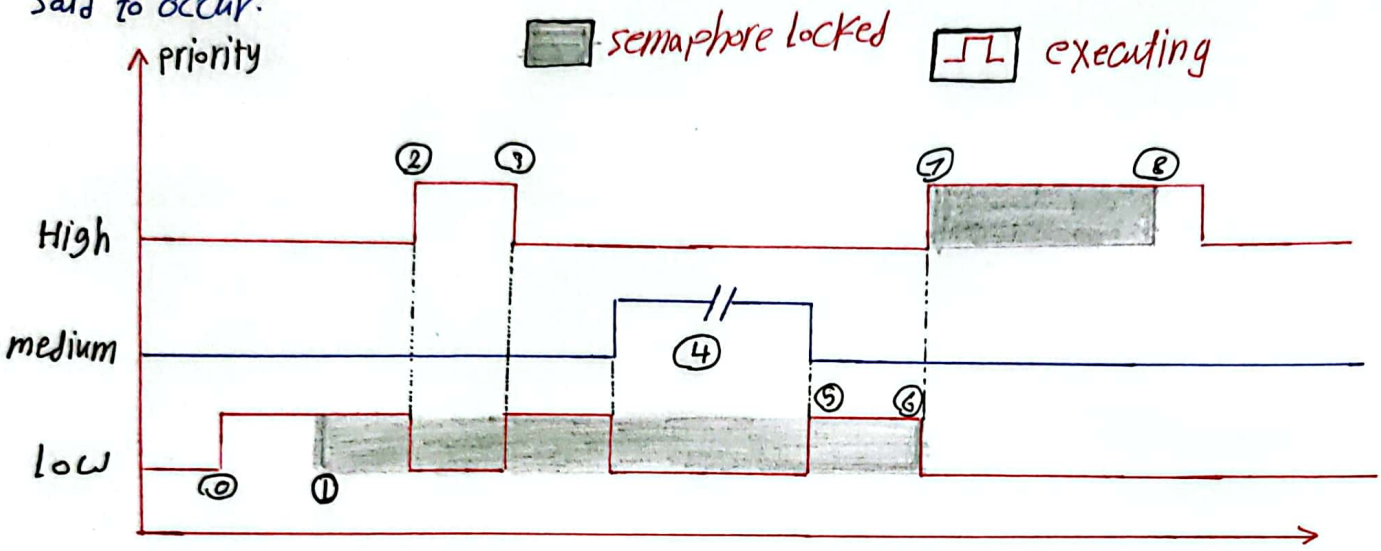
→ Dynamic priorities are used to avoid priority inversion.

→ in FreeRTOS : Lower the priority value means lower the priority.

* priority inversion :-

→ priority inversion is a scenario where a lower-priority task holds a shared resource needed by a higher-priority task.

→ this situation causes the higher-priority task to wait, here the priority inversion is said to occur.



0- a Low priority task starts to run and then enters a critical section by taking a mutex or semaphore.

1- a Low priority task takes a semaphore before being preempted by the High-priority task.

2- the High priority task preempts the Low-priority task and starts to run.

3- the High-priority task attempts to take the semaphore but it can't because it is still being held by the Low-priority task. So the High priority task enters the blocked state to wait for the semaphore to become available. So the scheduler returns execution to the Low-priority task to finish doing its job in the critical section.

→ this is where the term priority inversion comes into play. the lower priority task is running when the higher priority task asks should be the one that's running.

4- before releasing the semaphore by the Low-priority task, the medium-priority task preempts the Low-priority task and starts to run.

→ since the medium-priority task doesn't depend on the semaphore, it runs till complete.

5- the scheduler returns execution to the low priority task.

6-7- the low priority task finishes its job, and releasing the semaphore causes the High priority task to exit the blocked state as the semaphore holder.

8- when the High-priority task has finished with the semaphore it gives it back.

* 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.

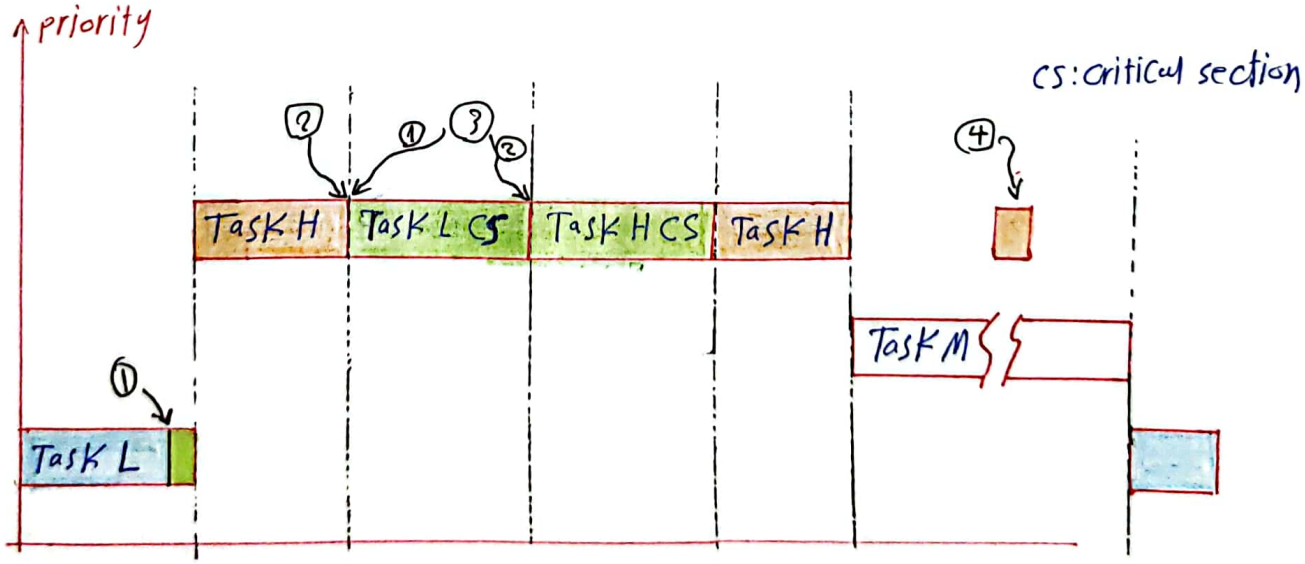
3- in critical ~~section~~ systems, prolonged priority inversion can Lead to failures.

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

* the ways to solve or reduce the priority inversion:-

1- priority inheritance :

→ it involves boosting the priority of a task holding a lock to that of any other task that tries to take the lock



① Task L takes Lock ② task H is blocked trying to take lock

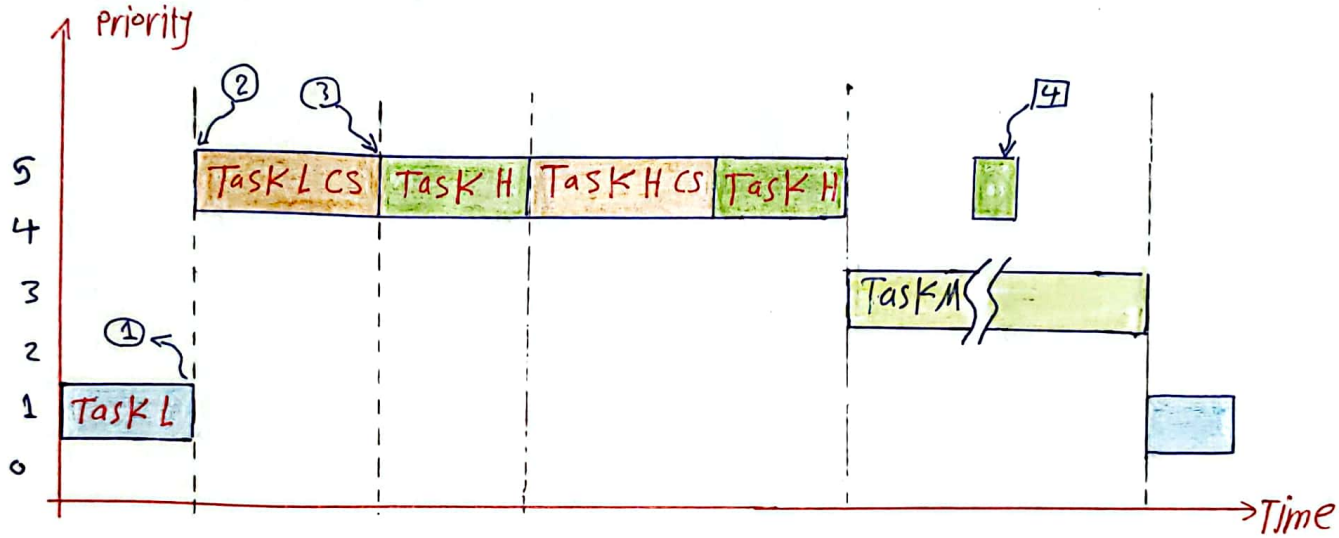
③-② the priority of task L is boosted to the priority of task H

④-② task L releases Lock ④ task H Can now freely preempt task M

→ Task L takes the lock. only when Task H attempts to take the lock, the priority of Task L boosted to that of Task H's.
→ Task M can no longer interrupt Task L until both tasks are finished in the critical section.
→ Task L's priority is dropped back to its original level once it releases the lock.
→ a scheduler must support dynamic priority scheduling.

2- Priority Ceiling :-

→ this protocol involves assigning a priority ceiling level to each resource or lock, which is a priority equal to the highest priority of any task which may lock the resource
→ whenever a task works with a particular resource or takes a lock, the task priority level is automatically boosted to the maximum priority of any task that needs to use the resource or lock.



→ Lock's priority ceiling = 5
→ whenever Task L takes the lock, its priority is boosted to 5 so that it will run at the same priority as task H. this prevents task M from running until task L and H are done with lock.

1- task L takes Lock
2- Task L executes at elevated priority
3- task L releases Lock
4- task H can now freely preempt task M

→ scheduler must support dynamic priority scheduling

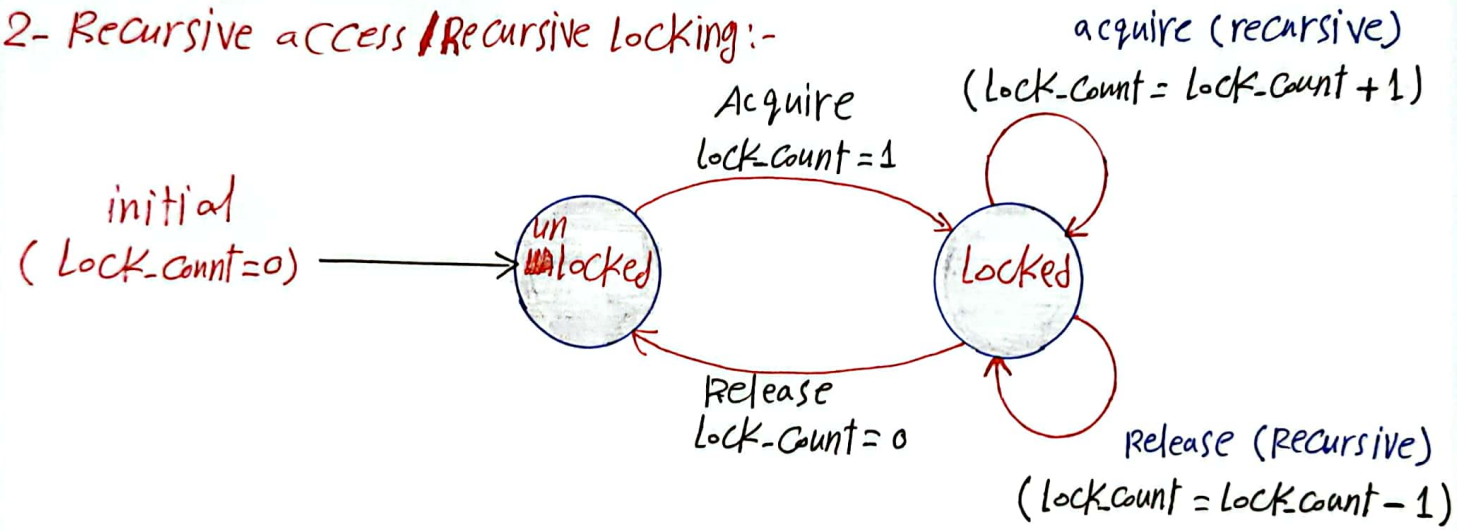
Mutex (Mutual EXclusion)

→ the mutex is a special type of semaphore binary used to protect access to a resource shared by more than one task.

→ Mutex supports additional features not found in binary or counting semaphores.
→ these key differentiating features include :-
- ownership
- Recursive access
- priority inversion avoidance protocols.
- Task deletion safety.

1- ownership
→ ownership of mutex is gained when a task first locks the mutex by acquiring it.
→ conversely, a task loses the ownership of the mutex when it unlocks it.
→ when a task owns the mutex, it is not possible for any task to lock or unlock that mutex
→ the binary semaphore can be released by any task, even a task that did not originally acquire the semaphore.

2- Recursive access / Recursive locking :-



→ Recursive locking allows a task to acquire a mutex multiple times while it is already locked
→ this feature is most useful when a task requiring exclusive access to a shared resource calls one or more routines that also require access to the same resource
→ A recursive mutex keeps track of the number of times it has been recursively locked by the owning task.
→ it prevents deadlock situations and ensures that the mutex is properly unlocked the same number of times it was locked.
→ the count mechanism used for the mutex tracks the number of times that the task owning the mutex has locked or unlocked the mutex. and this count is always unbounded, which allows multiple recursive access
→ the count used for the counting semaphore tracks the number of tokens that have been acquired or released by any task.

3- Task Deletion safety:-

→ this feature ensures that a task cannot be deleted while it is holding a mutex.

→ it is achieved by using task deletion locks in combination with mutex operations.

→ When a task acquires a mutex, it is protected from being untimely deleted until it releases the mutex.

4- priority inversion Avoidance

→ Mutex can avoid priority inversion by using protocols such as priority inheritance and ceiling priority

→ these protocols ensure that the priority of the lower priority task is raised to that of the higher priority task when inversion occurs.

---

* Semaphore APIs :-

→ Binary semaphore creation :-

*Defining a semaphore:-

→ A semaphore is referenced by a global variable of type xSemaphoreHandle

* creating a binary semaphore :-

SemaphorHandle_t xSemaphoreCreateBinary (void);

→ FreeRTOS provides this function to create a binary semaphore.

→ this function dynamically allocates the memory required for the semaphore.

→ it returns semaphoreHandle_t value, this handle is a reference to the newly created semaphore

→ configSupport_Dynamic_Allocation must be set to 1 in FreeRTOSConfig.h

→ Each binary semaphore requires a small amount of RAM to hold the semaphore's state.

→ if a binary semaphore is created using the previous function, the required RAM is automatically allocated from the FreeRTOS heap.

→ if a binary semaphore is created using xSemaphoreCreateBinaryStatic (), the RAM is provided by the application writer.

*Return Value:

→ Null : it means the semaphore could not be created due to insufficient heap memory.

→ any other value : it indicates that the semaphore has been created successfully the returned value should be stored as the handle to the created semaphore.

* xSemaphoreTake() API :-

BaseType_t xSemaphoreTake(SemaphoreHandle_t xSemaphore, TickType_t x-);

→ it takes a semaphore that has previously been created.

→ this function must not be used from an ISR.

* Parameters :-

→ xSemaphore :

→ the semaphore to be taken

→ a semaphore is referenced by a variable of type SemaphoreHandle_t.

→ it must be explicitly created before it can be used.

→ xTicksToWait :

→ the maximum time the task waits in the blocked state for the semaphore to become available

→ if it set to zero, the function returns immediately if the semaphore is unavailable.

* Return values :

pdPASS : the semaphore was successfully obtained.

pdFAIL :

→ the semaphore was not obtained.

→ the task might have waited for the semaphore, but the block time expired before it became available.

→ xSemaphoreTake(), should only be called from an executing task, not during scheduler initialization.

→ it must not be called within a critical section or while the scheduler is suspended.

---

* xSemaphoreGive() API :-

BaseType_t xSemaphoreGive(SemaphoreHandle_t xSemaphore);

→ it gives or releases a semaphore that has previously been created.

→ the semaphore must have been obtained earlier using xsemaphoreTake() if its initial value is available.

→ this function must not be called from an interrupt service routine(ISR). instead, use xSemaphoreGiveFromISR() for that purpose.

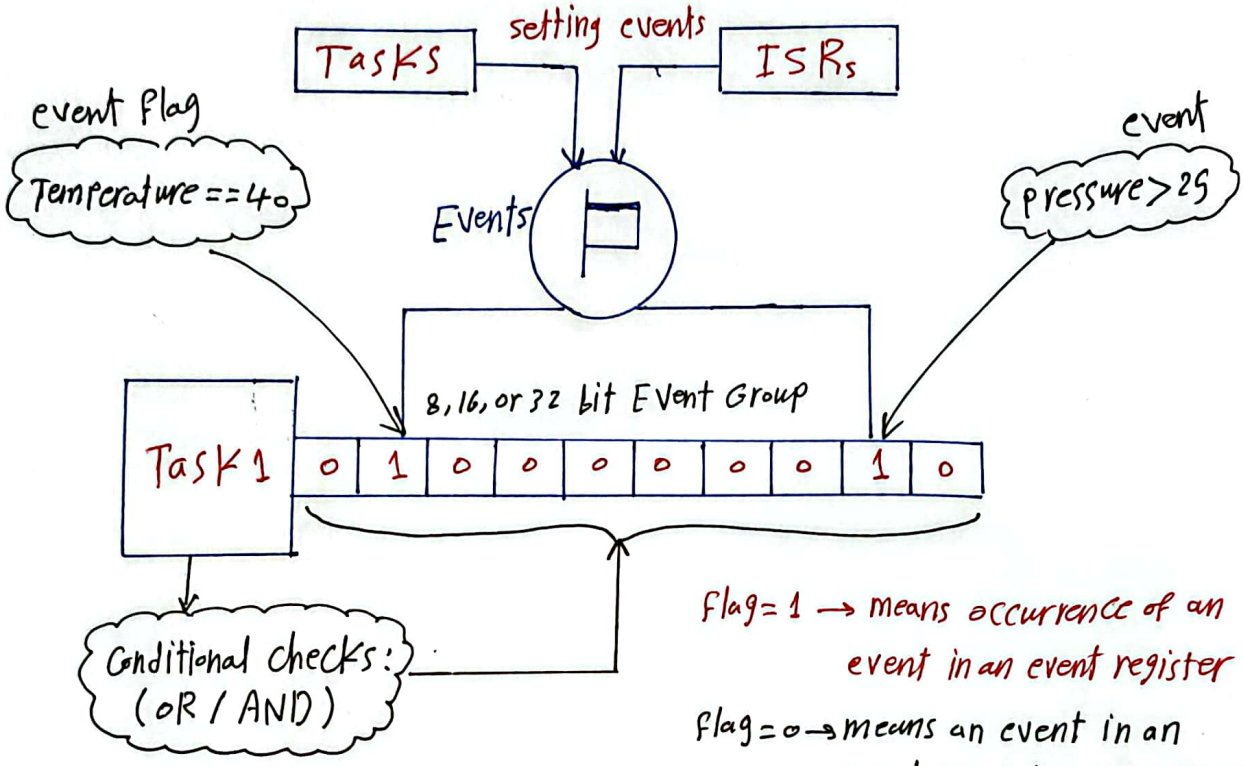
→ this function must also not be used on semaphores created using xSemaphoreCreateRecursiveMutex().

→ this function requires the same prameter and returns the same value of take API.

Inter-task event synchronization

→ Event group and event flags:

→ An event flag is a boolean value that represents the status of a single event within an event group.



→ an event flag is a single bit, like a binary semaphore

→ the single flag can be used to communicate the occurrence of one event to a task.

→ A task can be blocked (in waiting state) waiting for occurrence of one or more events to occur.

→ Event Group: is a special type of variable constructed from Event Flags, and stored in variables of type EventBits_t.

→ the number of bits (or flags) implemented within an event group is:

→ if configUSE_16_BIT_TICKS is set to 1 ⟶ 8 bits

→ if " " " " " is set to 0 ⟶ 24 bits

* Event Group APIs:-

→ the event group APIs provide functions to create, manipulate, and interact with event groups and their individual bits

→ setting event flags from an ISR is performed by sending messages from the ISR to a kernal task, which can reduce the need for extended critical sections within the code.

* APIs :-

→ xEventGroupCreate() : Creats a new event group
→ xEventGroupDelete() : Deletes an existing event group
→ xEventGroupGetBits() : Retrieves the current state of the event group's bits.
→ xEventGroupSetBits() : sets one or more bits in the event group, representing events that have occurred.
→ xEventGroupClearBits() : Clears one or more bits in the event group.
→ xEventGroupWaitBits() : Allows a task to wait for one or more bits to be set in the event group.

* Uses of Event Groups :-

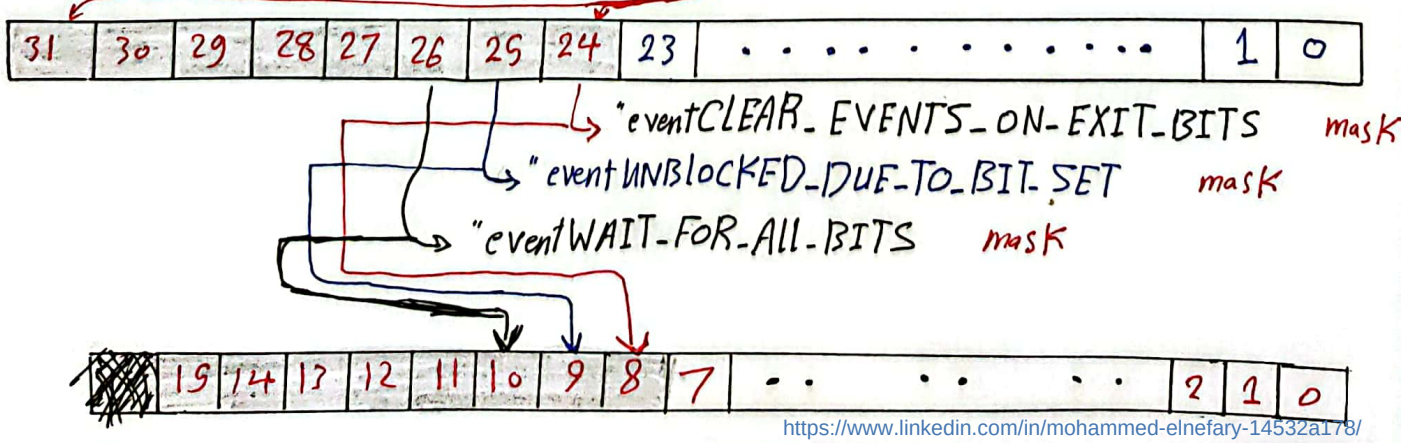
→ Task blocking : tasks can be blocked (not consuming any CPU time) while waiting for a single event flag or a combination of event flags within an event group.

→ Replacing Binary semaphores : An event group can replace multiple Binary semaphores, reducing processing and memory requirements.
  → a task can unblock when a single, a combination, or all event flags in the event group are set.

→ Task synchronization : Event Group can be used to synchronize tasks by keeping them in the blocked state until all participating tasks reach their synchronization point.
  → this synchronization is signaled by setting individual bits within the event group.

* the most significant 8-bits of the variable (event flag) in FreeRTOS are called "control Bits" or "Mask"

→ these bits have specific roles in controlling how tasks interact with event groups. they are defined by "mask" → "eventEVENT_BITS_CONTROL_BYTES"



- Bit 26 : (AND / OR)
  - → if bit 26 = 1 : the kernal will wait for all specified bits to be set
  - → if bit 26 = 0 : the kernal will wait for any one of the specified bits to be set.
- Bit 25 :
  - → this bit is set to 1 if any of the events we are waiting for occurs, it is dependent on the state of bit 26 to determine whether we are waiting for a single event or multiple events to change the task's state from blocked state to ready state.
  - → if the task's state changes without any of the specified events occurring, it indicates a timeout has happened.
- Bit 24 :
  - → if the bits we are waiting on occur and the task exits the blocked state to the ready state, and "bit 24 = 1", the kernal will clear all the bits that the task was waiting on.

→ event groups are stored in variables of type EventBits_t.

→ the number of bits (or flags) implemented within an event group depends on whether configUSE_16_BIT_TICKS or configTICK_TYPE_WIDTH_IN_BITS is used to control the type of TickType_t

→ the number of bits/flags implemented within an event group is 8 if configUSE_16_BIT_TICKS is set to 1, or 24 if it set to 0.

OR

8 → configTICK_TYPE_WIDTH_IN_BITS = TICK_TYPE_WIDTH_16_BITS

24 → 〃 〃 〃 〃 〃 = TICK_TYPE_WIDTH_32_BITS

56 → 〃 〃 〃 〃 〃 = TICK_TYPE_WIDTH_64_BITS

Inter Task Communication (Queue)

→ Message Queues :

→ a message queue is a buffer-like object through which tasks and ISRs send and receive messages to communicate and synchronize with data.

→ a message queue is like a pipeline.

→ it temporarily holds messages from a sender (source) until the intended receiver (sink) is ready to read them

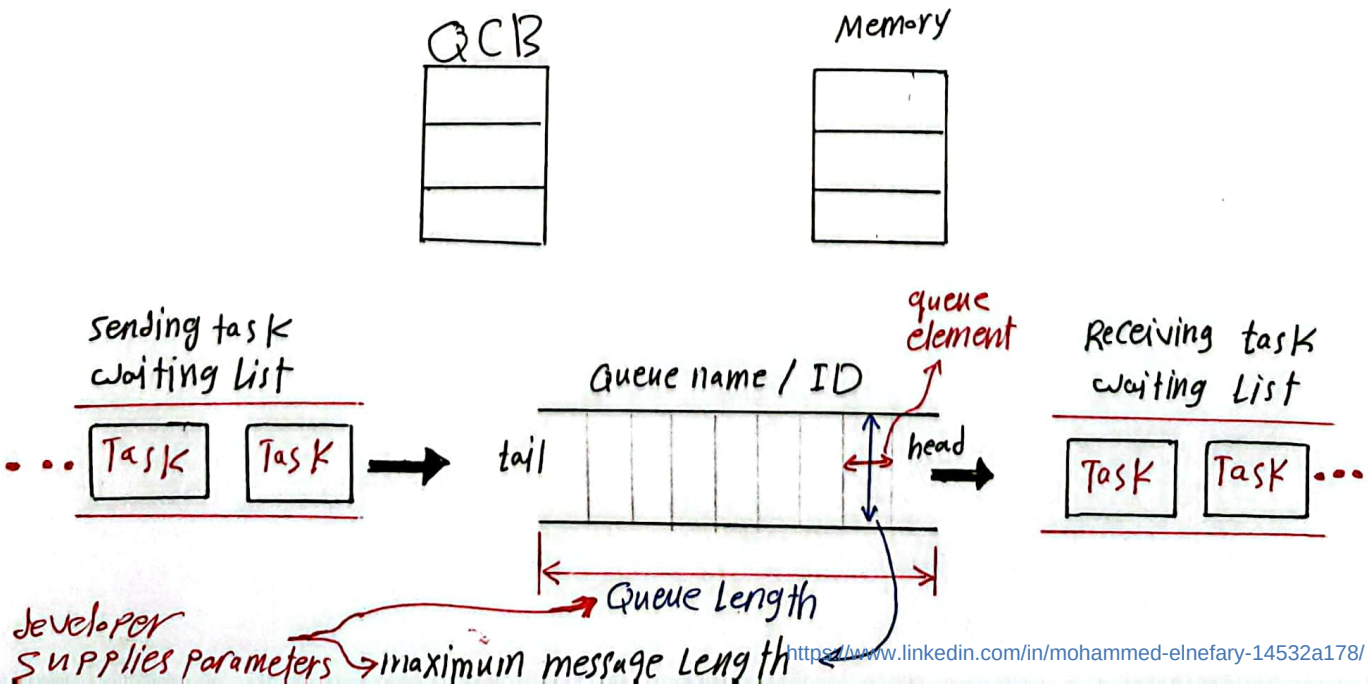
→ the temporary buffering allows the sending and receiving tasks to work independently. it means the tasks don't have to send and receive messages at the same time.

→ the buffer acts as a middle-man, holding the messages temporarily. this way, the sending tasks can simply deposit the message in the buffer, without having to wait for the receiving task to be ready. and the receiving task can retrieve the message from the buffer, without having to wory about the sending task's timing.

→ A message queue has several associated components that the kernal uses to manage the queue.

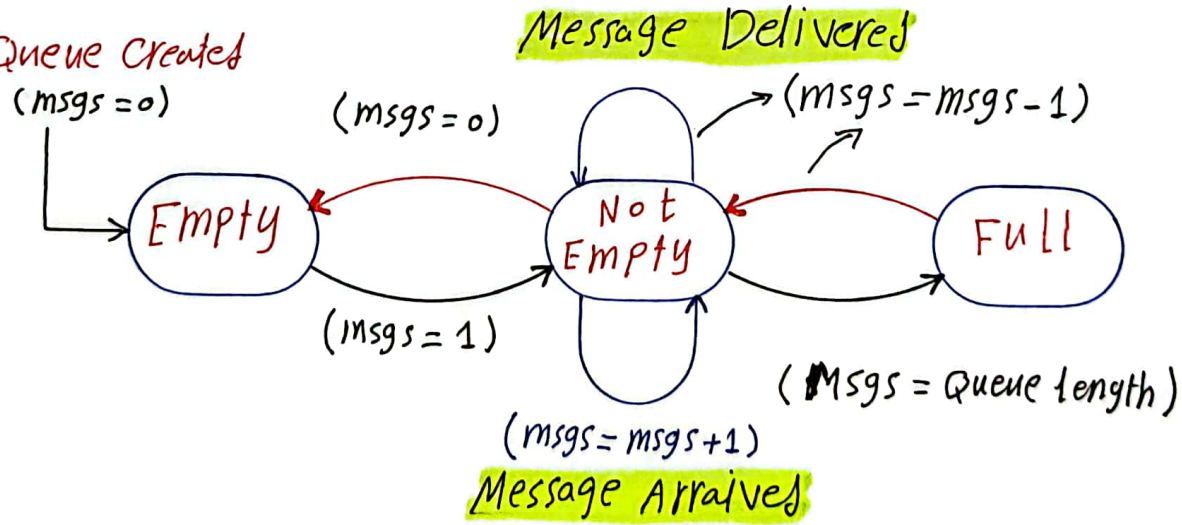
→ When a message queue is first created, it is assigned an associated :

1- Queue Control Block (QCB)
2- unique ID
3- message queue name
4- Memory buffer
5- Queue length
6- maximum message length
7- one or more task-waiting list



→ it is the kernal's job to assign a unique ID to a message queue and to create its QCB and task-waiting list.
→ the kernal also takes developer-supplies parameters, such as the length of the queue and the maximum message length to determine how much memory is required for the message queue.
→ After the kernal has this information, it allocates memory for the message queue from either a pool of system memory or some private memory space.
→ the message queue itself consists of a number of elements, each of them can hold a single message.
→ the elements holding the first and last messages are called the head and tail.
→ A message queue has two associated task-waiting lists:
  → the receiving task-waiting list consists of tasks that wait on the queue when it is empty
  → the sending list consists of tasks that wait on the queue when it is full.

* Message Queue states :.
→ As with other kernal objects, message queues follow the logic of a simple FSM (finite-state machine).



→ when a message queue is first created, the FSM is in the empty state.
→ if a task attempts to receive messages from this message queue while the queue is empty, the task blocks, and if the task chooses to be added to a waiting list for that message queue, it will be handled depending on the order of the waiting list, in either:
  → FIFO → priority-based order

→ if another task sends a message to the message queue, the message is delivered directly to the blocked task.

→ the blocked task is then removed from the task-waiting list and moved to either the ready or the running state. the message queue in this case remains empty because it has successfully delivered the message.

→ if another message is sent to the same message queue and no tasks are waiting in the message queue's task-waiting list, the message queue's state becomes not empty.

→ As additional messages arrive at the queue, the queue eventually fills up until it has exhausted its free space. At this point, the number of messages in the queue is equal to the queue's length, and the message queue's state becomes full.

→ while a message queue is in this state, any task sending messages to it will not be successful unless some other tasks request a message from that queue, thus freeing a queue element.

→ in some kernal implementations, when a task attempts to send a message to a full message queue, the sending function returns an error code to that task.

→ other kernal implementations allow such a task to block, moving the blocked task into the sending task-waiting list, which is separate from the receiving task-waiting list.

→ message queues can be used to send and receive a variety of data.

→ some of these messages can be quite long and may exceed the maximum message length, which is determined when the queue is created.

  → one way to overcome the limit on message length is to send a pointer to the data, rather than the data itself.

  → even if a long message might fit into the queue, it is sometimes better to send a pointer instead in order to improve both performance and memory utilization.

→ when a task sends a message to another task, the message normally is copied twice

- the first time, the message is copied when the message is sent from the sending task's memory area to the message queue's memory area
- the second copy occurs when the message is copied from the message queue's memory area to the receiving task's memory area.

* Typical Message queue operations include the following:.
  1- creating and deleting message queues
  2- sending and receiving messages
  3- obtaining messages queue information.

* Creating and deleting message queues:
→ When created, message queues are treated as global objects and are not owned by any particular task, typically, the queue to be used by each group of tasks or ISRs is assigned in the design.
→ When creating a message queue, a developer needs to make some initial decisions about the length of the message queue, the maximum size of the messages it can handle, and the waiting order for tasks when they block on a message queue.
→ Deleting a message queue automatically unblocks waiting tasks.
→ the blocking call in each of these tasks returns with an error. messages that were queued are lost when the queue is deleted.

→ API to create a queue:-

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

→ xQueueCreate() creates a new queue and returns a handle by which the queue can be referenced.
→ the first parameter → How many items should the queue hold?
→ the second parameter → What is the size of a single item in bytes?
→ returns:
  → if the queue is created successfully then a handle to the created queue is returned. if the memory required to create the queue could not be allocated then Null is returned.

→ API to delete a queue:-

```
Void vQueueDelete ( QueueHandle_t xQueue);
```

→ it deletes a queue → freeing all the memory allocated for storing of items placed on the queue
→ parameter → xQueue → a handle to the queue to be deleted.

* sending and receiving messages :-

→ when sending messages, a kernal typically fills a message queue from head to tail in FIFO or LIFO order

→ if a task attempts to send a message to a full queue, it can be blocked until space becomes available in the queue

→ this blocking can be configured with different policies, such as blocking forever, blocking for a timeout period, or not blocking

→ similar to sending: tasks can receive messages with the same blocking policies as they use for sending:

→ Not blocking: the task will just try to get a message and continue if none is available

→ Blocking with a timeout: the task will wait for a message, but only for a certain amount of time

→ Blocking forever: the task will wait indefinitely until a message becomes available.

→ when tasks try to receive messages from an empty queue, they get added to a waiting list

→ this waiting list can fill up if messages are being added to the queue slower than they are being taken out.

→ Conversely, the waiting list only starts filling up when the message queue is completely empty.

→ Messages can be read from the head of a message queue in two different ways:

1- Destructive read:

→ when a task successfully receives a message from a queue, the task permanently removes the message from the message queue's storage buffer.

2- Non-destructive read (not supported by all kernal implementations):

→ A receiving task peeks at the message at the head of the queue without removing it

→ it is used in applications that require at least once delivery assurances, as the message can be processed again.

→ APIs

xQueueSend()	xQueueSendFromISR()
xQueueSendToBack()	xQueueSendToBackFromISR()

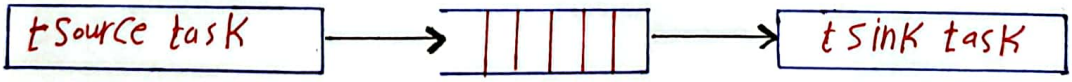
xQueueSendToFront() xQueueSendToFrontFromISR()
xQueueReceive() xQueueReceiveFromISR()

---

* obtaining Message Queue information :-
→ Different kernels allow developers to obtain different types of information about a message queue, such as :
  → the message queue ID → number of messages queued
  → the queuing order used for blocked tasks (FIFO or priority-based)
→ some calls might even allow developers to get a full list of messages that have been queued up.
→ the information is dynamic and might have changed by the time it's viewed.
→ these types of calls should only be used for debugging purposes.

Typical Message Queue use:
→ Non-interlocked, one-way data communication.
→ Interlocked, one-way data communication.
→ Interlocked, two-way data communication.
→ Broadcast Communication.

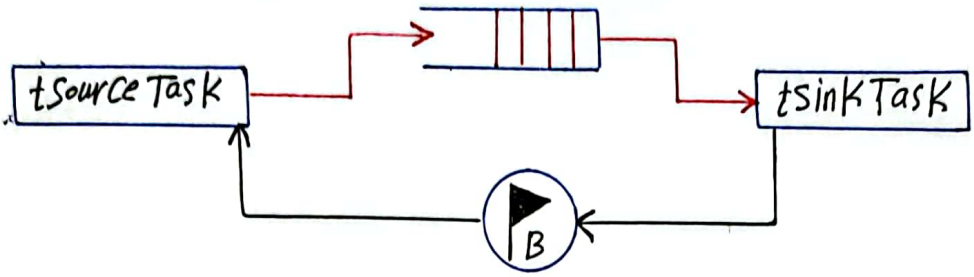
1-Non-Interlocked, one-way data communication :-



→ the simplest scenarios for message-based communication.
→ the activities of tSourceTask and tSinkTask are not synchronized.
→ tSourceTask simply sends a message, while it does not require acknowledgement from tSinkTask.
→ IF tSinkTask is set to a higher priority :-
  → it runs first until it blocks on an empty message queue.
  → As soon as tSourceTask sends the message to the queue, tSinkTask receives the message and starts to execute again.
→ IF tSinkTask is set to a lower priority :-
  → tSourceTask fills the message queue with messages.
  → Eventually, tSourceTask can be made to block when sending a message to a full message queue.
  → this action makes tSinkTask wakeup and start taking messages out of the queue.

- When ISRs send messages to the message queue, the must do so in a non-blocking way.
- if the message queue becomes full, any additional messages that the ISR sends to the message queue are lost.

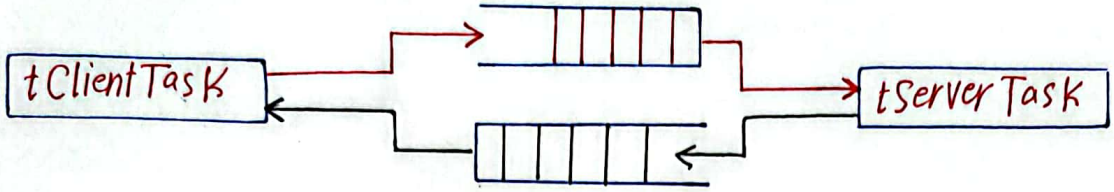
- Interlocked, one-way data communication :-



→ in some designs, a sending task might require a handshake (acknowledgement) that the receiving task has been successful in receiving the message.
→ this process is called interlocked communication, in which the sending task sends a message and waits to see if the message is received.
→ this process can be useful for reliable communications or task synchronization.
→ if the message for some reason is not received correctly, the sending task can resend it.
→ a binary semaphore initially set to 0 and a message queue with a length of 1.
→ this method is also called a mailbox.
→ tSourceTask sends message and blocks on the binary semaphore
→ tSinkTask receives the message and increments the binary semaphore
→ the semaphore wakes up tSourceTask.
→ tSourceTask executes and sends another message, blocking again..... loop.
→ pseudo code for Interlocked, one way data communication.

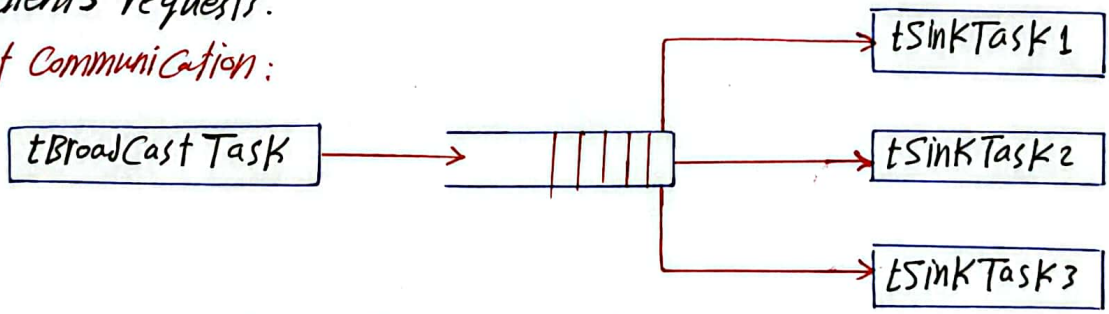
```
tSourceTask(){
    send message to message queue
    Acquire binary semaphore
}
tSinkTask(){
    Receive message from message queue
    Give binary semaphore
}
```

3- Interlocked, Two-way data Communication



→ Sometimes data must flow bidirectional between tasks, which is called interlocked, two-way data communication (Full-Duplex or tightly Coupled communication).
→ this form of communication can be useful when designing a client/server-based system.
→ tClientTask sends a request to tserverTask via a message queue
→ tServerTask fulfills that request by sending a message back to tClientTask.
→ tServerTask is set to a higher priority, allowing it to quickly fulfill client requests.
→ if multiple clients need to be set up, all clients can use the client message queue to post requests, while tserverTask uses a separate message queue to fulfill the different clients' requests.

4- Broadcast Communication:



→ Some message-queue implementations allow developers to broadcast a copy of the same message to multiple tasks.
→ Message broadcasting is a one-to-many-task relationship.
→ tBroadCastTask sends the message on which multiple tSinkTask are waiting.
→ in this scenario, tSinkTask 1, 2, and 3 have all made calls to block on the broadcast message queue, waiting for a message
→ when tBroadCastTask executes, it sends one message to the message queue, resulting in unblocking all three waiting tasks.
→ Note: not all message queue implementations might support the broadcasting facility.

Hook Functions

→ Hooks are callback functions supported by the FreeRTOS core, which could help with FreeRTOS fault handling.

→ these functions are user-defined functions which provide a mechanism for developer to add application-specific functionality without modifying the core FreeRTOS code.

→ each hook function should meet the following requirements:

- short as possible
- it can't be blocking, this means no delays, no wait functions are allowed to be used inside.

→ Hooks are disabled by default and should be enabled within the FreeRTOSConfig.h file.

→ Hook functions are defined in the FreeRTOS.c file with "weak" attribute, so we can overwrite them in our code without any issues.

* types of hooks:

1- Idle hook: called within the Idle task, at the end of its execution.
2- Tick hook: called within SisTick function
3- Malloc failed hook: called in case of memory allocation issues.
4- Stack overflow hook: called in case of issues with the stack of the OS.

1- Idle Hook:-

* Idle Task Characteristics:-

→ it is portTASK_FUNCTION() function within task.c file.
→ it has the lowest possible priority.
→ it can share same priority with other tasks.
→ it is performing the following operation (in endless loop):.

- check for deleted tasks to clean the memory.
- taskYIELD() if we are not using preemption (configUSE_PREEMPTION = 0)
- Get yield if there is another task waiting and we set (configIDLE_SHOULD_YIELD = 1)
- Executes vApplicationIdleHook() → if configUSE_IDLE_Hook = 1.
- perform low power entrance if configUSE_TICKLESS_IDLE != 0

→ **Idel Hook :-**

→ it is called from the context of the IDLE task which is automatically created by scheduler within osKernalStart() function.

→ the configUSE_IDLE_Hook must be set to 1 in FreeRTOSConfig.h file to get it called.

→ the idle hook is executed every iteration of the idle task's Loop.

→ the idle hook is primarily used during the development phase. it allows us to check how often the idle task is executed.

→ even the idle hook is enabled, it could not be executed if other conditions happens before calling the idle hook. it runs only if there are no tasks in ready state.

→ the idle hook function is typically represented by vApplicationIdleHook(void); which is a user defined function that can be added to our 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.

→ Do not use blocking functions (osDelay(),....) in this function or while(1)

---

* **Tick Hook :-**

→ every time the systick interrupt is trigger the tick' hook is called.

→ it is possible use tick hook for periodic events like watchdog refresh

- Be careful because the priority of the SysTick is the lowest priority in the system, so it could happen that the OS can be frozen for some time and the WDG will not be refreshed on time.

→ Enabled via the configUSE_TICK_Hook configuration option.

→ the tick hook function is called from an interrupt context, meaning it's executed directly when the SysTick interrupt occurs.

→ Because the tick hook is called within an interrupt context, it's crucial to avoid any blocking operations. and we should only use interrupt-safe FreeRTOS API functions (those ending in FromISR()).

- xPortSysTickHandler(); → xTaskIncrementTick();
  (configUSE_TICK_Hook == 1) ← check
  True → vApplicationTickHook();

* **Malloc Failed Hook Function:-**

→ this callback is called if the memory allocation process fails
  → pvPortMalloc() returns Null.

→ Defining malloc failed hook will help to identify problems caused by lack of heap memory.

→ Malloc failed hook will only get called if configUSE_MALLOC_FAILED_HOOK is set to 1 in FreeRTOSConfig.h. When it is set, an application must provide hook function with the following prototype:-

```
void vApplicationMallocFailedHook(void);
```

→ Do not use blocking functions in this function or while(1).

→ it is useful for logging memory allocation issues or triggering a system reset.

* **Stack overflow Hook:-**

→ FreeRTOS is able to check stack against overflow

→ stack overflow protection runtime check of stack "High Watermark"
  - During task creation, its stack memory space is filled with 0XA5 data.
  - During run time we can check how much stack is used by task-stack high water mark

→ configUSE_TRACE_FACILITY should be defined to 1

→ INCLUDE_uxTaskGetStackHighWaterMark should be defined to 1

→ configCHECK_FOR_STACK_OVERFLOW should be defined to 1

→ Prototype:-

```
uxTaskGetStackHighWaterMark(xTaskHandle xTask);
```

→ after call it with task handle as an argument returns the minimum amount of remaining stack for xTask is presented (Null means task which is currently in RUN mode)

→ High watermark: is the amount of stack that remained unused when the task-stack was at its greatest value.

* FreeRTOS offers two distinct methods for stack overflow detection:

1- Option1:

→ Stack can reach its deepest value after the RTOS kernal has swapped the task out of the running state because this is when the stack will contain the task context. At this point RTOS kernal can check whether stack pointer remains within valid stack space

→ stack overflow hook is called, if the stack pointer contains value outside

of the valid stack range.

→ this method is quick, but it can't guarantee catching all stack overflows.

→ To use this option only we need to set config CHECK_FOR_STACK_OVERFLOW to 1 within FreeRTOSConfig.h file or select option 1 within CHECK_FOR_STACK_OVERFLOW option when using CubeMX.

2- option 2 :-

→ when task is first created, its stack is filled with a known value (0xA5)

→ when moving the task from run mode, kernal can check last 20 bytes within valid stack range to ensure that these known values have not been over written.

→ stack overflow hook function is called if any of these 20 bytes not remain at their initial value

→ this method is less efficient than method one, but still fast

→ it is very likely to catch stack overflows but is still not guarantees to catch all overflows

→ to use this method in combination with option 1 set configCHECK_FOR_STACK_OVERFLOW to 2 (this is not possible to use only this option)

→ stack overflow hook function prototype :-

vApplicationStackOverflowHook(xTaskHandle *pxTask, signed char *pcName)

→ Do not use blocking functions or while(1) in this function.

→ To collect runtime statics of os components, there is dedicated function :-

→ osThreadList()

→ this function is calling vTaskList() within FreeRTOS API and is collecting information about all tasks and put them to the table.

Software Timer

→ software timers are used to schedule the execution of a function at a set time in the future, or periodically with a fixed frequency.

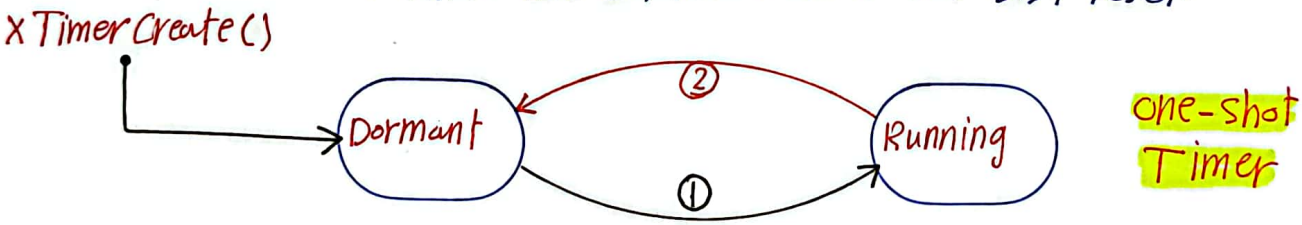
→ the function executed by the software timer is called the software timer's callback function.

→ software timers are implemented and managed by the FreeRTOS kernal. they do not require hardware support and are unrelated to hardware timers.

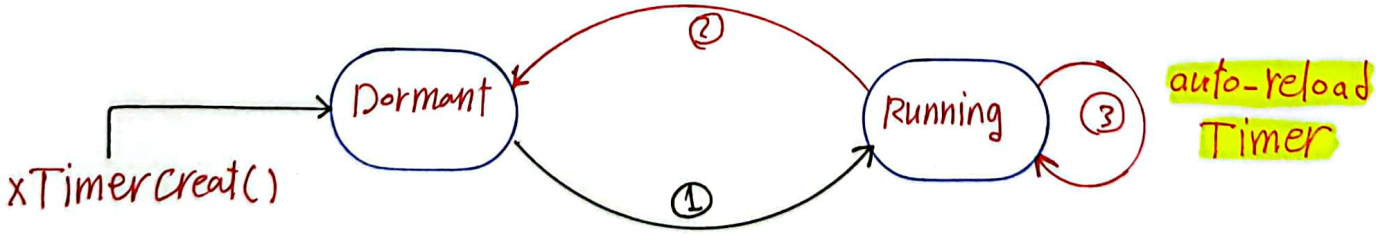
→ software timers donot consume processing time unless a software timer callback function is actively executing.

*Software Timer states :-

- Dormant state : a dormant software timer exists, and can be referenced by its handle, but is not running, so its callback function will not execute.
- Running state : a running software timer will execute its callback function after a time equal to its period, which has elapsed since the software timer entered the running state, or since the software timer was last reset.



1 : xTimerStart() xTimerReset() xTimerChangePeriod()
2 : xTimerStop() Timer expired / execute callbacks



1 : xTimerStart() xTimerReset() xTimerChangePeriod()
2 : xTimerStop()
3 : Timer expired / execute callback

* Software Timer Types :-

1- one-shot timers :

→ once started, a one shot timer will execute its callback function once only

→ a one-shot timer can be restarted manually but will not restart itself

2- Auto-reload timers :

→ once started, an auto-reload timer will restart itself each time it expires, resulting in periodic execution of its callback function.

* the RTOS Daemon (Timer service) Task :-

→ All software timer callback functions execute in the context of the same RTOS Daemon

→ the daemon task is a standard FreeRTOS task that is created automatically when the scheduler is started.

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

→ the daemon task is scheduled like any other FreeRTOS task; it will only process commands, or execute timer callback function

→ its priority and stack size are set by the configTIMER_TASK_PRIORITY and configTIMER_TASK_STACK_DEPTH compile time configuration constants respectively. Both constants are defined within FreeRTOSConfig.h.

→ the scheduler also creats automatically a message queue used to send commands to the timers task (timer start, timer stop ....)

→ the number of elements of this queue (number of massages that can be hold) are configurable through the define :- configTIMER_QUEUE_LENGTH

* Software Timer Configuration with FreeRTOS :

→ configUSE_TIMERS :

0 : disabled, no timer service task

1 : includes software timers functionality and automatically creats timer service task on scheduler start

→ configTIMER_TASK_PRIORITY

→ priority for timer service task from the range between IDLE task priority and configMAX_PRIORITY - 1 .

→ configTIMER_QUEUE_LENGTH
  → this set the maximum number of unprocessed commands that the timer command queue can hold at any one time.
→ configTIMER_TASK_STACK_DEPTH
  → sets the size of the stack (in words) allocated to the timer service task.

* Timer Creation :-

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

→ once a timer has been created it needs to be started and managed by the timer APIs :-

```
BaseType_t xTimerStart ( TimerHandle_t xTimer, TickType_t xTickToWait);
BaseType_t xTimerStop ( TimerHandle_t xTimer, TickType_t xTickToWait );
BaseType_t xTimeReset (  "          "          "          " );
BaseType_t xTimerDelete(  "          "          "          " );
```

The

End