FreeRTOS Documentation

# 1)Scheduling Schemes

**a) Co-Operative Scheduli**ng([Program](https://github.com/sumitadep002/Embedded_Systems/tree/master/RTOS/workspace/01_Co_Operative)):

Cooperative scheduling, also known as cooperative multitasking or cooperative multitasking scheduling, is a type of process or task scheduling used in computer operating systems. In cooperative scheduling, the operating system relies on the voluntary cooperation of processes or tasks to relinquish control of the CPU, allowing other processes to execute. It is in contrast to preemptive scheduling, where the operating system can forcibly interrupt a running process to allocate CPU time to another process based on predefined scheduling policies.

Here's how cooperative scheduling works:

1. \*\*Voluntary Yielding\*\*: In a cooperative scheduling environment, each running process or task is responsible for yielding control of the CPU voluntarily. A process can do this by explicitly invoking a yield or release mechanism provided by the operating system or by some other means of cooperation.

2. \*\*No Forced Preemption\*\*: The operating system does not forcibly preempt a process. It relies on the processes to give up the CPU willingly, which means that if a process misbehaves or enters an infinite loop, it can potentially monopolize the CPU.

3. \*\*Simple Synchronization\*\*: Since processes must cooperate to share CPU time, they often use synchronization mechanisms like semaphores or mutexes to coordinate their activities and avoid conflicts.

4. \*\*Low Overhead\*\*: Cooperative scheduling typically has lower overhead compared to preemptive scheduling because there is no need for the OS to constantly monitor and interrupt running processes. However, it also relies on processes to be well-behaved and to yield control fairly.

5. \*\*Predictable Execution\*\*: In cooperative scheduling, processes can have more predictable execution times because they are not unexpectedly preempted by the operating system. This can be an advantage in certain real-time or embedded systems.

However, cooperative scheduling has some drawbacks:

- \*\*Susceptibility to Misbehaving Processes\*\*: If a process doesn't cooperate by yielding control when needed, it can lead to system freezes or slow performance.

- \*\*Limited Responsiveness\*\*: Cooperative scheduling can be less responsive in systems with a mix of well-behaved and misbehaved processes, as misbehaved processes can block others from executing.

- \*\*Difficult Debugging\*\*: Debugging cooperative scheduling systems can be challenging, as issues related to process cooperation and synchronization can be hard to diagnose.

Cooperative scheduling is less common in modern general-purpose operating systems because preemptive scheduling is more robust and can handle misbehaving processes better. However, it is still used in some specific scenarios where predictability and low overhead are more important than robustness, such as certain embedded systems and real-time applications.

**b) Preemptive scheduling** ([Program](https://github.com/sumitadep002/Embedded_Systems/tree/master/RTOS/workspace/01_Pre_Emptive_Scheduling))

Preemptive scheduling is a type of process or task scheduling used in computer operating systems where the operating system can forcibly interrupt or preempt a running process and allocate CPU time to another process based on predefined scheduling policies. In preemptive scheduling, the OS constantly monitors the state of running processes and makes decisions about when to switch between them. This is in contrast to cooperative scheduling, where processes voluntarily yield control of the CPU.

Here's how preemptive scheduling works:

1. \*\*Time Slicing\*\*: The CPU time is divided into small time slices, often referred to as time quantum or time slice. Each process is allocated a time slice during which it can execute. When a time slice expires, the operating system can preempt the current running process, even if it hasn't completed its task.

2. \*\*Priority-Based Scheduling\*\*: Preemptive scheduling systems often use priority levels to determine which process gets CPU time. Processes with higher priority levels are given precedence over those with lower priorities. The operating system can dynamically adjust priorities based on various factors.

3. \*\*Interrupt Mechanisms\*\*: Preemptive scheduling relies on hardware and software interrupt mechanisms. Hardware interrupts can be generated by external events, like a keypress or a timer, prompting the OS to switch to a different process. Software interrupts, also known as system calls or exceptions, can be used to request services from the operating system.

4. \*\*Context Switching\*\*: When a process is preempted, the operating system performs a context switch. This involves saving the state of the currently running process, such as its CPU registers and program counter, and loading the saved state of the new process that will be executed. Context switching incurs some overhead, but it allows for efficient multitasking.

Benefits of preemptive scheduling:

1. \*\*Fairness\*\*: Preemptive scheduling ensures that no single process can monopolize the CPU for an extended period, promoting fairness and equal opportunity for all processes.

2. \*\*Responsiveness\*\*: The system remains responsive to user input and external events because the OS can quickly switch to a process that needs to be executed.

3. \*\*Prioritization\*\*: Important and time-sensitive tasks can be given higher priority, ensuring they get CPU time when needed.

4. \*\*Multi-Processing Support\*\*: Preemptive scheduling is essential for multi-processing environments, where multiple CPUs or cores are available, as it enables efficient utilization of all available resources.

However, preemptive scheduling also has some potential drawbacks:

1. \*\*Overhead\*\*: The frequent context switches between processes can introduce overhead, especially in situations where many processes are competing for CPU time.

2. \*\*Complexity\*\*: Managing priorities and dealing with potential race conditions and synchronization issues can make preemptive scheduling more complex to implement and debug.

In modern general-purpose operating systems, preemptive scheduling is the predominant method because it provides better control over system resources, responsiveness, and fairness, and can handle misbehaving or uncooperative processes more effectively.

# 2)RTOS Tick

In a Real-Time Operating System (RTOS), a "tick" is a fixed time interval that serves as the basis for scheduling tasks, setting timers, and handling real-time events. It helps maintain predictable and deterministic behavior in the system by dividing time into discrete increments, allowing tasks to run in specific intervals, and enabling precise timekeeping.

It can be configured in **FreeRTOSConfig.h** by modifying **configTICK\_RATE\_HZ** MACRO

# 3)Task Handle

TaskHandle\_t is a data type commonly used in FreeRTOS, an open-source real-time operating system. It represents a handle or reference to a task, allowing you to manage and interact with tasks in a FreeRTOS application. It is typically used for creating, deleting, suspending, resuming, or communicating with tasks. In short, TaskHandle\_t is a way to identify and manipulate tasks within FreeRTOS.

# 4)Task Handler

`void Task\_Handler(void \*param) ` is a common function prototype for a task in an embedded system using an RTOS like FreeRTOS. This function is responsible for defining the behavior of a task. The `void \*param` parameter allows you to pass data or configuration information to the task. Inside ` Task\_Handler `, you implement the specific functionality of the task, and it runs concurrently with other tasks in the system.

# 5)xTaskCreate()

In an RTOS like FreeRTOS, `xTaskCreate` is a function used to create a new task. It allows you to define a new task and specify its properties, such as its entry function and priority. Here's an explanation of its parameters and return value:

\*\*Parameters\*\*:

1. `pxTaskCode`: This is a pointer to the task function that will be executed when the task is created. It's of type `TaskFunction\_t`, which is essentially a function pointer to the task code. This is where you define the behavior of the task.

2. `pcName`: A human-readable name or identifier for the task. This is useful for debugging and identifying tasks in the system.

3. `usStackDepth`: The size of the task's stack in words. This determines the amount of memory allocated for the task's stack. The unit of measurement depends on the architecture and compiler but is typically in bytes.

4. `pvParameters`: A pointer to parameters or data that you want to pass to the task's entry function (`pxTaskCode`). It's of type `void \*`, so you can pass any data type by casting it appropriately.

5. `uxPriority`: The priority of the task. Tasks with higher priorities get CPU time over tasks with lower priorities. This value is usually between 0 (lowest priority) and `configMAX\_PRIORITIES - 1` (highest priority).

6. `pxCreatedTask`: A pointer to a `TaskHandle\_t` variable that will hold the handle to the newly created task. This is an output parameter and allows you to reference the task after it's created.

\*\*Return Value\*\*:

The `xTaskCreate` function returns a value of type `BaseType\_t`. It typically returns `pdPASS` if the task was successfully created, indicating success. If there was an issue with task creation, it returns an error code like `pdFAIL`. You should check this return value to handle any potential errors during task creation.

For example, you might use `xTaskCreate` like this:

TaskHandle\_t myTaskHandle;

BaseType\_t result = xTaskCreate(TaskFunction, "TaskName", 100, (void \*)parameters, 1, &myTaskHandle);

if (result == pdPASS) {

// Task created successfully.

} else {

// Handle task creation error.

}

```

In this example, the `TaskFunction` is the task's entry function, "TaskName" is the task's name, 100 is the stack depth, `(void \*) parameters` is the data to be passed to the task, 1 is the task priority, and `myTaskHandle` will hold the handle to the created task.

6)vTaskStartScheduler()

`vTaskStartScheduler` is a function in the FreeRTOS real-time operating system that initializes and starts the task scheduler. When called, it sets up the multitasking environment, including creating and managing the idle task, and begins scheduling tasks to execute. It's typically the last step in configuring and starting an RTOS application and is crucial for task execution to begin.

7) taskYield()

TaskYield() is a function in many real-time operating systems (RTOS) that allows a task to voluntarily yield control of the CPU. When a task calls `taskYield()`, it temporarily gives up its execution time, allowing the RTOS to switch to another ready task, if one is available with the same or higher priority. This function is often used to ensure fair CPU time allocation in multitasking environments and to improve system responsiveness.

# 8)FromISR() APIs

"FromISR" functions in real-time operating systems are designed to safely perform certain task-related operations from interrupt service routines (ISRs). They ensure proper synchronization and minimize the impact on task scheduling and data integrity when ISRs interact with tasks and the RTOS.

# 9)vTaskDelay()

`vTaskDelay()` is a function in FreeRTOS that allows a task to voluntarily delay its execution for a specified number of system ticks. It is commonly used for adding time delays and controlling task timing in a real-time operating system.

# 10Idle Task ([Program](https://github.com/sumitadep002/Embedded_Systems/tree/master/RTOS/workspace/05_Idle_Hook))

The "Idle Task" is a special task in an operating system, including real-time operating systems like FreeRTOS. Its primary purpose is to execute when no other tasks are ready to run. In essence, it's a placeholder task that prevents the CPU from idling when there's no productive work to do. It typically consumes minimal CPU resources and might perform low-priority background tasks, such as managing the system's idle time or power-saving features.

# 11)SysTick Handler

The SysTick handler is a component of the ARM Cortex-M microcontroller architecture.

It is a system timer that generates periodic interrupts.

The SysTick timer can be used for various purposes, such as scheduling tasks in an RTOS, measuring time intervals, and more.

When a SysTick interrupt occurs, the processor can execute a specific interrupt service routine (ISR) to perform actions based on the timer's periodicity.

In the context of an RTOS like FreeRTOS, the SysTick handler is often used to trigger context switches, which allow different tasks to run in a cooperative multitasking environment.

The SysTick handler is part of the Cortex-M core, and its behavior can be configured and customized based on the microcontroller's capabilities and the specific application's needs.

# 12) PendSV (Pendable Service) Handler:

* The PendSV handler is another component of the ARM Cortex-M architecture, specifically designed to facilitate context switching in RTOS environments.
* It's a special-purpose interrupt that can be triggered from software to request a context switch between tasks.
* The PendSV handler is generally used to initiate context switches in preemptive RTOSs, allowing higher-priority tasks to preempt lower-priority ones.
* By triggering the PendSV interrupt, the RTOS can initiate a context switch to run the highest-priority ready task.
* In an RTOS like FreeRTOS, the PendSV handler plays a vital role in managing task scheduling, ensuring that the highest-priority ready task runs when necessary.
* The PendSV handler is part of the Cortex-M core and is used in conjunction with other RTOS mechanisms to achieve preemptive multitasking.
* In summary, the SysTick handler is a system timer that generates periodic interrupts, and it's often used in RTOS environments for scheduling and timekeeping. The PendSV handler is a specialized interrupt used to request context switches in RTOSs, allowing for preemptive multitasking by ensuring that higher-priority tasks can interrupt lower-priority ones.

# 13)Application Hook Functions([Program](https://github.com/sumitadep002/Embedded_Systems/tree/master/RTOS/workspace/05_Idle_Hook))

Application Hook Functions in an RTOS like FreeRTOS are callback functions that allow you to customize and extend the behavior of the RTOS at specific points in its execution. These hooks are often provided by the RTOS for you to define your own code, which gets executed in response to certain events or situations. They are used for a variety of purposes, including system monitoring, debugging, and fine-tuning.

The Idle Hook Function, often referred to as **vApplicationIdleHook**, is one of these application hook functions. It is called by the RTOS whenever the CPU is in an idle state, meaning there are no tasks ready to execute. The primary purpose of the Idle Hook Function is to perform background, low-priority tasks or enter a low-power mode to save energy when the system is not busy. It's an ideal place to place code for system status monitoring, power management, or other non-critical tasks that can be executed during idle periods.

# 14)xTaskNotify([Program](https://github.com/sumitadep002/Embedded_Systems/tree/master/RTOS/workspace/03_Task_Notfication))

`xTaskNotify()` is a function in FreeRTOS, a real-time operating system, used to send notifications from one task to another. It provides a way for tasks to communicate or synchronize by signaling events or sharing information.

\*\*Parameters\*\*:

1. `xTaskToNotify`: This is the handle of the task that you want to notify. The notification will be sent to this task. It is of type `TaskHandle\_t`.

2. `ulValue`: This is an unsigned long value that you can pass along with the notification. It's a 32-bit value that carries information or data associated with the notification.

3. `eAction`: This parameter specifies the action to take when notifying the task. You can choose from options like `eSetBits`, `eIncrement`, or `eSetValueWithOverwrite`. These actions determine how the `ulValue` is processed.

`xTaskNotify()` allows tasks to efficiently signal events and pass data between them, aiding in inter-task communication and synchronization in a real-time system.

# 15)xTaskNotifyWait([Program](https://github.com/sumitadep002/Embedded_Systems/tree/master/RTOS/workspace/03_Task_Notfication))

`xTaskNotifyWait()` is a function in FreeRTOS used by a task to wait for a notification from another task. It allows a task to block until it receives a notification or a specified period of time elapses.

\*\*Parameters\*\*:

1. `ulBitsToClearOnEntry`: This is an unsigned long value that specifies which notification bits to clear when the task enters the blocked state. These bits can be cleared from a task's notification value to indicate that specific conditions have been met.

2. `ulBitsToClearOnExit`: Similarly, this parameter specifies which notification bits to clear when the task exits the blocked state. This is typically used to indicate that the task has processed the notification.

3. `pulNotificationValue`: This is a pointer to an unsigned long variable where the notification value will be stored. When the task receives a notification, the value is copied to this variable, allowing the task to access the data associated with the notification.

4. `xTicksToWait`: It specifies the maximum amount of time the task should remain in the blocked state, waiting for a notification. This is the timeout value in ticks. If set to zero, the function returns immediately; if set to `portMAX\_DELAY`, it waits indefinitely.

`xTaskNotifyWait()` is a key mechanism for tasks to efficiently synchronize and communicate by waiting for notifications with optional timeout capabilities, which makes it useful for managing real-time tasks.

# 16)semaphore

In the context of FreeRTOS, a semaphore is a synchronization mechanism that helps control access to shared resources and manage task synchronization within the FreeRTOS real-time operating system. FreeRTOS provides several semaphore types, including binary semaphores and counting semaphores.

- Binary Semaphore([Program](https://github.com/sumitadep002/Embedded_Systems/tree/master/RTOS/workspace/07_Semaphore)): In FreeRTOS, a binary semaphore has two states: taken and not taken. It is often used for mutual exclusion and signaling between two tasks. One task takes the semaphore (sets it to "taken"), and another task can give it back (sets it to "not taken"). This is useful for protecting critical sections of code or for signaling between tasks.

- Counting Semaphore([Program](https://github.com/sumitadep002/Embedded_Systems/tree/master/RTOS/workspace/09_Semaphore_Counting)): A counting semaphore, also available in FreeRTOS, allows multiple tasks to access a shared resource concurrently, up to a specified limit. Tasks can take the semaphore and release it when done. This is useful for managing access to a finite pool of resources, like memory buffers or I/O channels.

FreeRTOS provides API functions for creating, taking, and giving semaphores, allowing tasks to synchronize and share resources in a controlled manner. Semaphores play a critical role in coordinating the execution of tasks and preventing race conditions in FreeRTOS applications.

# 17)semaphore APIs

In FreeRTOS, semaphores are often manipulated using their handles, which are pointers to the semaphore structures. The handle allows tasks to interact with the semaphore without needing to access the underlying data directly. Here are some commonly used semaphore functions and how they work with semaphore handles:

1. Semaphore Creation: To create a semaphore, you typically use a function like `xSemaphoreCreateBinary()` or `xSemaphoreCreateCounting()`. These functions return a semaphore handle that you store in a variable for future use.

SemaphoreHandle\_t xSemaphore;

xSemaphore = xSemaphoreCreateBinary();

2. \*\*Semaphore Taking (Wait)\*\*: You use the `xSemaphoreTake()` function to request and take a semaphore. If the semaphore is available (not taken by other tasks), the function will return immediately. If not, the calling task will be blocked until the semaphore is given (freed) by another task.

if (xSemaphoreTake(xSemaphore, portMAX\_DELAY) == pdTRUE) {

// Successfully acquired the semaphore, execute the critical section.

}

3. \*\*Semaphore Giving (Signal)\*\*: To release a semaphore, you use the `xSemaphoreGive()` function. This allows other tasks that are waiting for the semaphore to proceed.

xSemaphoreGive(xSemaphore);

4. \*\*Deleting a Semaphore\*\*: You can use `vSemaphoreDelete()` to release the resources associated with a semaphore when it's no longer needed.

vSemaphoreDelete(xSemaphore);

These functions operate on the semaphore handle you obtained when creating the semaphore. The handle serves as an abstraction that allows tasks to interact with the semaphore without directly accessing the internal data structures, enhancing safety and encapsulation. Semaphores and their handles play a crucial role in FreeRTOS for tasks to coordinate access to shared resources and synchronize their execution.

# 18)Mutex & it’s APIs ([Program](https://github.com/sumitadep002/Embedded_Systems/tree/master/RTOS/workspace/11_Mutex))

In the context of FreeRTOS, a Mutex (short for mutual exclusion) is a synchronization mechanism used to protect shared resources from concurrent access by multiple tasks. A mutex ensures that only one task can access the protected resource at a time, preventing race conditions and data corruption. Here's how Mutexes work in FreeRTOS:

1. \*\*Creation\*\*: You can create a mutex using the `xSemaphoreCreateMutex()` function. This function returns a Mutex handle, which is essentially a pointer to a Mutex structure.

```c

SemaphoreHandle\_t xMutex;

xMutex = xSemaphoreCreateMutex();

```

2. \*\*Taking the Mutex\*\*: To access the shared resource, a task must first "take" the Mutex using the `xSemaphoreTake()` function. If the Mutex is not available (i.e., another task has already taken it), the calling task will be blocked until the Mutex is released by the other task.

if (xSemaphoreTake(xMutex, portMAX\_DELAY) == pdTRUE) {

// Critical section: Access the protected resource.

// Release the Mutex when done.

xSemaphoreGive(xMutex);

}

3. \*\*Releasing the Mutex\*\*: After a task finishes using the shared resource (the critical section), it should "give" the Mutex back to make it available for other tasks. This is done using the `xSemaphoreGive()` function.

xSemaphoreGive(xMutex);

4. \*\*Deleting the Mutex\*\*: When the Mutex is no longer needed, you can delete it using the `vSemaphoreDelete()` function to free up associated resources.

vSemaphoreDelete(xMutex);

Mutexes are commonly used in FreeRTOS to protect critical sections of code where shared resources, such as global variables, hardware peripherals, or data buffers, need exclusive access. They ensure that tasks cooperate and take turns when accessing these resources, preventing data corruption and conflicts.

In summary, Mutexes in FreeRTOS provide a way for tasks to synchronize and safely access shared resources, ensuring data integrity and avoiding concurrency issues.

# 19)Software Timers ([Proram](https://github.com/sumitadep002/Embedded_Systems/tree/master/RTOS/workspace/10_Timers))

In the context of FreeRTOS, software timers are a feature that allows you to create and manage timers within your real-time operating system. Software timers are used to execute specific actions or functions at specified intervals or after a certain delay, enabling you to schedule tasks to run periodically or after a time delay. Here's how software timers work in FreeRTOS:

1. \*\*Creating a Software Timer\*\*:

- You can create a software timer using the `xTimerCreate()` function, which returns a handle to the created timer.

TimerHandle\_t xTimer;

xTimer = xTimerCreate("MyTimer", pdMS\_TO\_TICKS(1000), pdTRUE, 0, vTimerCallback);

- In this example, "MyTimer" is the timer's name, `pdMS\_TO\_TICKS(1000)` specifies a 1000 ms (1-second) timer, `pdTRUE` makes the timer auto-reload, and `vTimerCallback` is the function to execute when the timer expires.

2. \*\*Starting and Stopping Timers\*\*:

- You can start a timer using `xTimerStart()`, and you can stop it using `xTimerStop()`.

xTimerStart(xTimer, 0);

xTimerStop(xTimer, 0);

3. \*\*Timer Callback Function\*\*:

- When the timer expires, the specified callback function (`vTimerCallback` in the creation example) is executed. This is where you define the task or function to run when the timer triggers.

void vTimerCallback(TimerHandle\_t xTimer) {

// Your code to be executed when the timer expires.

}

4. \*\*Deleting a Timer\*\*:

- When you're done with a timer, you can delete it using `xTimerDelete()` to release associated resources.

vTimerDelete(xTimer);

Software timers in FreeRTOS are useful for tasks that need to perform periodic actions, such as sampling sensors, updating display screens, or managing timeouts. They provide a way to schedule tasks in a cooperative multitasking environment, ensuring that time-critical operations occur at specified intervals or after specific time delays. Software timers help with task scheduling and synchronization in real-time systems.