What is RTOS and how it handles interruptions:

RTOS stands for Real-Time Operating System. It is an operating system designed to manage and control real-time applications and tasks in embedded systems and other applications where precise timing and responsiveness are crucial. Unlike general-purpose operating systems (such as Windows, Linux, or macOS), which prioritize multitasking and provide various services, an RTOS is optimized for deterministic behavior and predictable execution.

Key characteristics of RTOS:

1. Deterministic Timing.
2. Task Management.
3. Interrupt Handling.
4. Resource Management.
5. Preemption.
6. Synchronization and Communication.
7. Click and Timer Services.

RTOS (Real-Time Operating System) handles interruptions (also known as interrupts) through a combination of hardware and software mechanisms to ensure timely and predictable response to events. Here's how an RTOS typically handles interruptions:

1. **Interrupt Vector Table**: When an interrupt occurs, the CPU looks up the appropriate interrupt handler address in the interrupt vector table. This table contains pointers to the specific interrupt service routines (ISRs) for different types of interrupts.

2. **Priority Levels**: Interrupts are usually assigned priority levels. The RTOS scheduler ensures that higher-priority interrupts can preempt lower-priority ones. This ensures that critical tasks can be executed without delay.

3. **Interrupt Service Routines (ISRs)**: An ISR is a small piece of code that is executed in response to an interrupt. It performs the immediate processing required for the interrupt. ISRs are typically written in a way that minimizes their execution time, as they can block the normal execution of tasks.

4. **Interrupt Context and Task Context**: When an interrupt occurs, the CPU switches from executing a task's context to executing the ISR's context. The ISR runs in a special context that has limited memory space and execution time. This context switch introduces minimal delay in responding to the interrupt.

5. **Deferred Processing**: In some cases, the ISR might not perform all the required processing directly. Instead, it may defer some processing to a lower-priority task. This can help reduce the execution time of the ISR and improve system responsiveness.

6. **Interrupt Locking**: During certain critical sections of an ISR, it might be necessary to prevent other interrupts from occurring to ensure data consistency and integrity. This is typically achieved through interrupt locking mechanisms, such as disabling interrupts temporarily or using hardware features like interrupt masking.
7. **Interrupt Nesting**: Some RTOSs support interrupt nesting, where a higher-priority interrupt can interrupt the execution of a lower-priority interrupt. This allows for a more flexible handling of interrupts and ensures that the most critical tasks are addressed first.
8. **Interrupt Latency**: RTOSs strive to minimize interrupt latency—the time between the occurrence of an interrupt and the start of its associated ISR. Lower interrupt latency is crucial for meeting real-time requirements.
9. **Interrupt Queues and Priorities**: If multiple interrupts of the same type occur before the ISR completes its execution, the RTOS may use interrupt queues to manage the order in which they are serviced, based on their priorities.