What is RTOS

A Real Time Operating System, commonly known as an RTOS, is a software component that rapidly switches between tasks, giving the impression that multiple programs are being executed at the same time on a single processing core.

In actual fact the processing core can only execute one program at any one time, and what the RTOS is actually doing is rapidly switching between individual programming threads (or Tasks) to give the impression that multiple programs are executing simultaneously.

Scheduling Algorithms:

When switching between Tasks the RTOS has to choose the most appropriate task to load next. There are several scheduling algorithms available, including Round Robin, Co-operative and Hybrid scheduling. However, to provide a responsive system most RTOS's use a pre-emptive scheduling algorithm.

Tasks and Priorities:

In a pre-emptive system each Task is given an individual priority value. The faster the required response, the higher the priority level assigned. When working in pre-emptive mode, the task chosen to execute is the highest priority task that is able to execute. This results in a highly responsive system.

Interrupt handling:

Priority inversion

One of the pitfalls of interrupt handling is priority inversion, which occurs when a high-priority task is blocked by a low-priority task that holds a shared resource, such as a mutex or a semaphore. This can cause the high-priority task to miss its deadline, or to be preempted by a medium-priority task that is not blocked. To avoid priority inversion, you can use the priority inheritance protocol, which allows the low-priority task to temporarily inherit the priority of the high-priority task that is waiting for the resource. This way, the low-priority task can finish its critical section and release the resource faster, allowing the high-priority task to resume.

Stack overflow

Stack overflow is a common pitfall of interrupt handling, which happens when the stack memory allocated for an interrupt service routine (ISR) or a task is not enough to store the local variables, function calls, and context switches. This can cause data corruption, memory leaks, or unexpected behavior. To avoid stack overflow, you can use static analysis tools to estimate the worst-case stack usage of ISRs and tasks and allocate enough stack memory accordingly. Dynamic stack monitoring can detect and report stack overflows at runtime, allowing for appropriate actions like resetting the system or logging the error. Stack guards protect the boundaries of the stack memory from being overwritten by other memory regions. Additionally, you can use the __attribute__((naked)) directive to avoid saving and restoring the registers and the stack pointer in the ISR prologue and epilogue, opting instead to handle context switches with assembly code.

Coding errors

A fourth pitfall of interrupt handling is coding errors, which are mistakes or bugs in the ISR code that can cause unexpected results or system failures. These coding errors can be hard to debug and fix, especially if they occur sporadically or unpredictably. To avoid them, you should use simple and clear logic in your ISRs, avoiding complex calculations, loops, or branching statements. Furthermore, volatile keywords or memory barriers can be used to ensure that the compiler does not optimize away or reorder the accesses to shared variables or memory-mapped registers. Additionally, atomic operations or critical sections can be employed to protect the shared variables or resources from concurrent access by multiple ISRs or tasks. Lastly, debug tools or instrumentation can be used to trace and monitor the execution and performance of your ISRs, so as to identify any anomalies or bottlenecks. Interrupt handling is an essential skill for embedded software development with RTOS on ARM architecture; with a knowledge of the common pitfalls and appropriate solutions, you can enhance the quality and efficiency of your RTOS application.