5.5 The Core Challenge: ISRs and Tasks Synchronization

Understanding the Problem: Shared Data and Race Conditions

When a hardware interrupt occurs, the CPU immediately stops executing the current task and jumps to the ISR. This can happen at any time, even in the middle of a single line of C code that takes multiple machine instructions to execute. If the ISR and the task both access the same global variable, the system is vulnerable to a **race condition**.

A race condition is an undesirable situation that occurs when the outcome of a process depends on the uncontrollable sequence of events. In our case, it's a bug that occurs when the timing of the interrupt corrupts shared data.

A Classic Example of a Race Condition:
Imagine a global variable `volatile int counter = 0;`.

- An ISR, triggered by a timer, is programmed to increment the counter: `counter++`;.
- A task in the main application is programmed to decrement it: `counter--`;.

Let's trace a potential failure scenario. Assume `counter` is currently `10`.

1. **Task Executes:** The task reads the value of `counter` (10) into a CPU register.
2. **Task Calculates:** The CPU calculates the new value, `10 - 1 = 9`.
3. **CONTEXT SWITCH (INTERRUPT):** Before the task can write the value `9` back to the `counter` variable in memory, a hardware interrupt occurs!
4. **ISR Executes:** The ISR runs. It reads the value of `counter` from memory (which is still 10).
5. **ISR Calculates:** The ISR calculates `10 + 1 = 11`.
6. **ISR Writes:** The ISR writes the value `11` back to the counter variable in memory.
7. **ISR Finishes:** The interrupt is complete, and the CPU returns control to the task, restoring its state exactly where it left off.
8. **Task Resumes:** The task is completely unaware that it was interrupted. Its next step is to write its calculated value (`9`) back to the `counter` variable.
9. **Corruption:** The value `11` that the ISR correctly calculated is now overwritten with `9`. The increment operation has been completely lost.

This is the fundamental problem of concurrency: protecting shared resources from uncontrolled, simultaneous access.

The ISR Context and `...FromISR()` Functions

To solve the synchronization problem, we need tools to manage access to shared data. However, as we learned in Part 2, ISRs operate in a special **interrupt context**. They are not tasks and are not managed by the RTOS scheduler. This leads to a critical rule: **an ISR can never block**.

If an ISR tried to wait for a resource (like calling `xQueueSend()` and the queue was full), it would effectively block. But since the ISR is not a task, the scheduler has no other context to switch to. The entire system would freeze, leading to a crash.

To solve this, FreeRTOS provides a special set of ISR-safe functions that are non-blocking. You can recognize them by their `...FromISR()` suffix.

- `xQueueSend()` -> `xQueueSendFromISR()`
- `xSemaphoreGive()` -> `xSemaphoreGiveFromISR()`

These functions include an optional parameter, `pxHigherPriorityTaskWoken`. An ISR uses this parameter to inform the RTOS if its action (e.g., giving a semaphore) has unblocked a task that has a higher priority than the task that was originally interrupted. If so, the RTOS can perform an immediate context switch to the higher-priority task as soon as the ISR finishes, ensuring the system remains as responsive as possible.

Revision #1
Created 2025-08-28 12:57:46 UTC by GI
Updated 2025-08-28 13:02:43 UTC by GI