Introduction to Real-Time Operating Systems: Part 3

Call: +44 (0)23 8098 8890
E-mail: sales@itdev.co.uk (mailto:sales@itdev.co.uk)

Posted 19th October 2017, By James H (/company/about/team/james-h)

Introduction

In the embedded world, the use of a Real Time Operating System (RTOS) is commonplace, and with the advent of the Internet of Things (IoT) they are becoming more so. You might be deciding to use an RTOS for the first time, or perhaps you are thinking of moving to a new RTOS. This is the third article in our series on RTOSes to help you to get started.

The previous article explained the concept of pre-emption, why an RTOS differs from a general OS in this matter, and the reasons for the difference. It asked "what causes a lack of ability for one task to pre-empt another?" and concluded that there were three reasons.

- Lack of pre-emption support
- Interrupts being disabled (for long periods)
- Locks and issues like priority inversion (discussed in part 4 (/blog/introduction-to-real-time-operating-systems-part-4) of this series)

This article will deal with the second reason, namely interrupt disabling.

Disabling Interrupts

The time a system takes to service an interrupt, that is, the time between an interrupt being asserted and the first instruction of the associated interrupt service routine (ISR) being run, is known as the system's *interrupt latency.* An RTOS will generally have a far more fastidious attention to reducing this latency than a general OS.

Interrupt disabling inherently increases latency. If interrupts are disabled whilst an event occurs, the event cannot be serviced until interrupts are enabled again. If this event is critical, the latency introduced may be unacceptable.

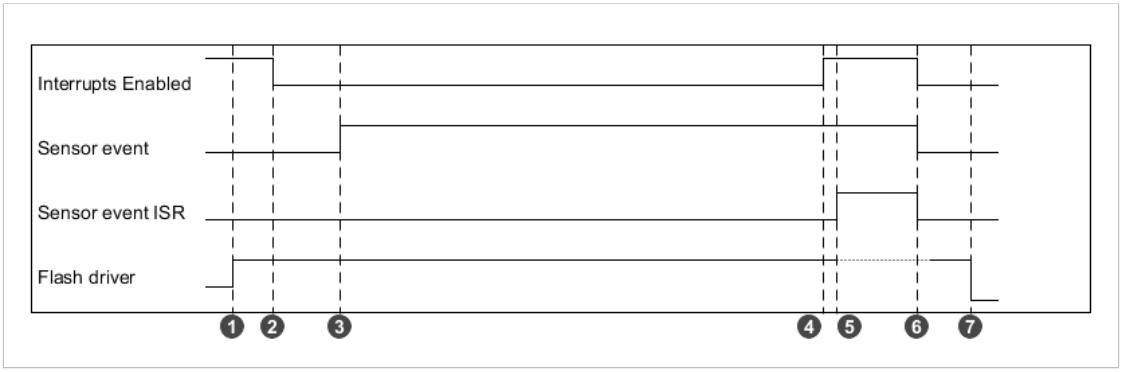
RTOSes are specifically designed to minimise interrupt latency. One of the ways in which they do this is through policy: the RTOS will use atomic primitives where possible to avoid disabling interrupts and will only disable interrupts where absolutely necessary and for as short a time as possible.

Another strategy for reducing latency is to carry out most of the interrupt processing outside of the interrupt context. This is something that PREEMPT-RT introduced to Linux:

> "Instead of running interrupt handlers in hard interrupt context, PREEMPT.RT runs them in kernel threads … The kernel thread becomes interruptible … you can put a low priority on non-important interrupts and a higher priority on important … tasks."[1]

The ISR routine will generally do as little as possible and will leave the real processing of the ISR to a task. The idea is that normally interrupts are not pre-emptible and so the actual ISR code should complete as quickly as possible to minimise system latency. Tasks, however, are pre-emptible, so by putting the bulk of the interrupt processing into a task allows the non-critical sections of an interrupt to be pre-empted by higher priority tasks and interrupts. Even if the hardware supports the nesting of interrupts they cannot be infinitely nested, and so the same applies. Additionally, interrupt or pre-emption disabling will also be strongly discouraged unless absolutely necessary.

We could imagine this in the context of a self-driving car. Storing sensor data is far less important than stopping the car when someone steps out in front of it, and we see below how disabling interrupts in a device driver can affect the system's response to more critical events. The following timing diagram shows what happens to our system when a badly written storage driver, which wants to do an atomic write of a block of data to the device, disables interrupts to ensure that it isn't pre-empted. I have actually seen a driver do this "out in the wild", although thankfully not in anything that would go anywhere near a safety critical system!



1. The storage driver begins running its write routine.
2. The storage driver disables interrupts to ensure the write is atomic.
3. Meanwhile, the sensor system detects that someone has stepped out in front of the car. The subsystem interrupts the CPU, but is held off because interrupts are still disabled.
4. The storage driver re-enables interrupts because it has finished doing its write.
5. The OS is now interrupted by the unmasked detection event and the sensors system's ISR is run. What?! Wait a minute! The car has detected that a person has stepped out in front of it, but it waited t_5-t_3 time periods before it did anything about it?! The time defined by $t_{latency} = t_5$-t_3 is the interrupt latency in this case and I'm sure we'd all agree it is completely unacceptable.

This, of course, is a pretty bad scenario and obviously contrived. First of all the sensor event would never be tied to an interrupt that can be masked and secondly one should never write a driver like this! The principle, however, is clearly shown. The RTOS has to respond to events as promptly as possible to meet its deadlines and to help ensure this, interrupt latency must be kept to a minimum.

Different RTOS designs have different approaches to task vs. interrupt priorities. Green Hill's INTEGRITY RTOS[2], for example, helps reduce latency by doing fancy things like letting task priorities and interrupt priorities overlap. This is handy because when a critical task is about to run, the RTOS will *automatically* disable lower priority interrupts when it schedules in the task. This, of course, relies on the hardware interrupt controller and hardware design being able to support this behaviour too. They also claim that interrupts are never disabled during system calls [3].

We've learnt about interrupt disabling and the effect it can have on a system. The next article will cover a barrier to correct pre-emption and adherence to priorities that are very particular to real-time systems, namely priority inversion.

References

1. E. Brown, "Real-time Linux explained, and contrasted with Xenomai and RTAI," 10 02 2017. [Online]. Available: http://linuxgizmos.com/real-time-linux-explained/ (http://linuxgizmos.com/real-time-linux-explained/).
2. https://www.ghs.com/products/rtos/integrity.html (https://www.ghs.com/products/rtos/integrity.html)
3. Green Hills Software, "Green Hills Software's INTEGRITY Real Time Operating System Available for Freescale's PowerQUICC III Integrated Communications Processors," 26 04 2004. [Online]. Available: https://www.ghs.com/news/20040426_processors.html (https://www.ghs.com/news/20040426_processors.html).

< prev (/blog/introduction-to-real-time-operating-systems-part-2) next > (/blog/introduction-to-real-time-operating-systems-part-4)

Latest Blog Posts