



Jack G. Ganssle

# Interrupt Latency

**My** dad was a mechanical engineer who spent his career designing spacecraft. I remember, in the early days of the space program, how he and his colleagues analyzed seemingly every aspect of their creations' behavior. Center of gravity calculations insured that the vehicles were always balanced. Thermal studies guaranteed nothing got too hot or too cold. Detailed structural mode analysis even identified how the system would vibrate, to avoid destructive resonances induced by the brutal launch phase. Though they were creating products that worked in a harsh and often unknown environment, their computations profiled how the systems would behave.

Think about civil engineers. Today no one builds a bridge without "doing the math" first. That delicate web of cables supporting a thin dancing roadway is simply going to work. Period. The calculations proved it long before contractors started pouring concrete.

Airplane designers also use quantitative methods to predict performance. When was the last time you heard of a new plane design that wouldn't fly? Yet wing shapes are complex and notoriously resistant to analytical methods. In the absence of adequate theory, the engineers rely on extensive tables acquired over decades of wind tunnel experiments. The engineers can still understand how their product will work—in general—before bending metal.

Compare this to our field. Despite decades of research, formal methods to prove software correctness are still impractical for real systems. We embedded engineers build then test, with no real proof that our products will work. When we pick a CPU, clock

speed, or memory size, we're betting that our off-the-cuff guesses will be adequate when, a year later, we're starting to test 100,000 or more lines of code. Experience plays an important role in getting the resource requirements right. All too often, luck is even more critical. But hope is our chief tool, as well as the knowledge that generally, with enough heroics, we can overcome most challenges.

**Knowledge is power, but often we don't know what's really happening in our systems. This column contains some tricks to help you find out.**

In my position as embedded gadfly looking into thousands of projects, I figure some 10% to 15% are total failures due to inadequate resources: the 8051 just can't handle that firehose of data; the PowerPC part was a good choice but the program grew to twice the size of available flash, and with the new cost model, the product is not viable.

Recently, I've been seeing quite a bit written about ways to make our embedded systems more predictable, to ensure that they react fast enough to external stimuli, to guarantee processes complete ontime. To my knowledge there is no realistically useful way to calculate predictability. In most cases we build the system and start changing stuff if it runs too slowly. Compared to aerospace and civil engineers, we're working in the dark.

It's especially hard to predict behavior when asynchronous activities alter program flow. Multitasking and inter-

rupts both lead to impossible-to-analyze problems. Recent threads on USENET, as well as some discussions at the Embedded Systems Conference, suggest banning interrupts altogether! I guess this does lead to a system that's easier to analyze, but it strikes me as too radical.

I've built polled systems. Yech. Worse are applications that must deal with several different things more or less concurrently, without using multitasking.

The software in both situations is invariably a convoluted mess. Twenty years ago, I naively built a steel thickness gauge without an RTOS, only to later have to shoehorn one in. Too many asynchronous things were happening; the in-line code grew to outlandish complexity. I'm still trying to figure out how to explain that particular sin to Saint Peter.

## Latency

A particularly vexing problem is to ensure the system will respond to external events in a timely manner. How can we guarantee that an interrupt will be recognized and processed fast enough to keep the system reliable?

Let's look in some detail at the first of the requirements: that an interrupt be recognized in time. Simple enough, it seems. Page through the processor's databook and you'll find a

spec called “latency,” a number always listed at sub-microsecond levels. No doubt a footnote defines latency as the longest time between when the interrupt occurs and when the CPU suspends the current processing context. That would seem to be the interrupt response time, but it ain’t.

Latency as defined by CPU vendors varies from zero (the processor is ready to handle an interrupt *right now*) to the max time specified. It’s a product of what sort of instruction is going on. It’s a bad idea to change contexts in the middle of executing an instruction, so the processor generally waits till the current instruction is complete before sampling the interrupt input. Now, if it’s doing a simple register-to-register move, which may be only a single clock cycle, which is a mere 50ns on a zero wait state 20MHz processor. Not much of a delay at all.

Other instructions are much slower. Multiplies can take dozens of clocks.

Read-modify-write instructions (such as “increment memory”) are also inherently pokey. Max latency numbers come from these slowest of instructions.

Many CPUs include looping constructs that can take hundreds, even thousands of microseconds. A block memory-to-memory transfer, for instance, initiated by a single instruction, might run for an awfully long time, driving latency figures out of sight. All processors I’m aware of will accept an interrupt in the middle of these long loops to keep interrupt response reasonable. The block move will be suspended, but enough context is saved to allow the transfer to resume when the interrupt service routine (ISR) completes.

So the latency figure in the datasheet tells us the longest time for which the processor can’t service interrupts. The number is useless to firmware engineers.

Okay, if you’re building an extreme cycle-countin’, nanosecond-poor, gray-

hair-inducing system, then perhaps that 300ns latency figure is indeed a critical part of your system’s performance. For the rest of us, real latency—the 99% component of interrupt response—comes not from what the CPU is doing, but from our own software design. And that’s hard to predict at design time. Without formal methods, we need empirical ways to manage latency.

If latency is time between getting an interrupt and entering the ISR, then surely most of it occurs because we’ve disabled interrupts. It’s because of the way we wrote the darn code. Turn interrupts off for even a few C statements and latency might run to hundreds of microseconds, far more than those handful of nanoseconds quoted by CPU vendors.

No matter how carefully you build the application, you’ll be turning interrupts off frequently. Even code that never issues a “disable interrupt”

instruction does, indeed, disable them often. Every time a hardware event issues an interrupt request, the processor itself does an automatic disable, one that stays in effect until you explicitly re-enable them inside of the ISR. Skyrocketing latency results.

On many processors we don't so much turn interrupts off as change priority levels. A Motorola 68000 receiving an interrupt on level five will prohibit all interrupts at this and lower levels until our code explicitly re-enables them in the ISR. Higher priority devices will still function, but latency for all level one to five devices is infinity until the code does its thing.

So, in an ISR, re-enable interrupts as soon as possible. When reading code, one of my "rules of thumb" is that code that does the enable just before the return is probably flawed. Most of us were taught to defer the interrupt enable until the end of the ISR. But that prolongs latency. Every other interrupt (at least at or below that priority level) will be shut down until the ISR completes. It's better to enter the routine, do all of the non-reentrant things (like handling hardware), and then enable interrupts. Run the rest of the ISR with interrupts on. You'll reduce latency and increase performance.

The downside is a need for more stack space if that same interrupt can re-invoke itself. There's nothing wrong with this in a properly designed and reentrant ISR, but the stack will grow until all pending interrupts get serviced.

The second biggest cause of latency is excessive use of the disable interrupts instruction. Shared resources—global variables, hardware, and so on—will cause erratic crashes when two asynchronous activities access them simultaneously. We have to keep the code reentrant by keeping all such accesses atomic, by limiting access to a single task at a time. The classic approach is to disable interrupts around such accesses. Though a simple solution, it comes at the cost of increased latency. See the April 2001 issue's Beginner's Corner ("Reentrancy," p. 183) for more on managing reentrancy.

## Taking data

So what is the latency of your system? Do you know? Why not? It's appalling that so many of us build systems with an "if the stupid thing works at all, ship it" philosophy. It seems to me there are certain critical parameters we must understand in order to properly develop and maintain a product. Questions you should ask are: is there any free ROM space? Is the system 20% loaded or 99%? How bad is the max latency?

Latency is pretty easy to measure. Simply instrument each ISR with an instruction that toggles a parallel output bit high when the routine starts. Drive it low just as it exits. Connect this bit to one input of an oscilloscope, tying the other input to the interrupt signal itself.

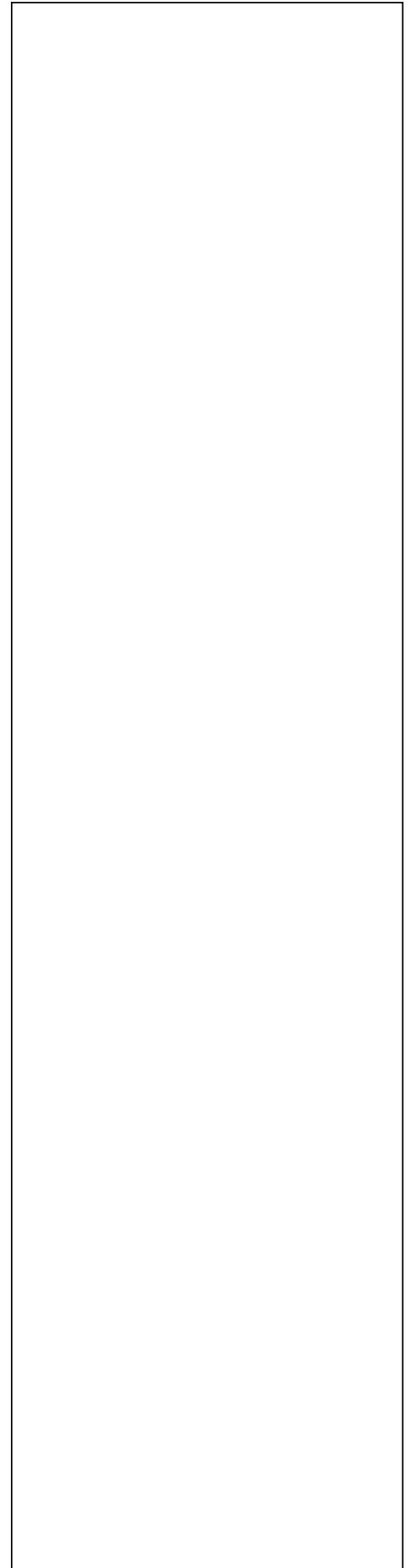
This simple setup produces a breathtaking amount of information. Measure time from the assertion of the interrupt till the parallel bit goes high. That's latency, minus a bit for managing the instrumentation bit. Twiddle the scope's time base to measure this to any level of precision required.

The time the bit stays high is the ISR's total execution time. Tired of guessing how fast your code runs? This is quantitative, accurate, and cheap.

In a real system, interrupts happen frequently. Latency varies depending on what else is going on. Use a digital scope in storage mode. After the assertion of the interrupt input, you'll see a clear space. That's the minimum system latency to this input. Then there will be hash, a blur as the instrumentation bit goes high at different times relative to the interrupt input. These represent variations in latency. When the blur resolves itself into a solid high, that's the maximum latency. All this, for the cost of one unused parallel bit.

If you've got a spare timer channel, there's another approach, which requires neither extra bits nor a scope. Build an ISR just for measurement purposes that services interrupts from the timer.

On initialization, start the timer counting up, and program it to inter-



rupt when the count overflows. Have it count as fast as possible. Keep the ISR dead simple, with minimal overhead. This is a good thing to write in assembly language to minimize unneeded code. Too many C compilers push everything inside interrupt handlers.

The ISR itself reads the timer's count register and sums the number into a long variable, perhaps called `total_time`. Also increment a counter (iterations). Clean up and return.

The trick here is that although the timer reads zero when it tosses out the overflow interrupt, the timer register continues counting even as the CPU is busy getting ready to invoke the ISR. If the system is busy processing another interrupt, or perhaps stuck in an interrupt-disabled state, the counter continues to increment. An infinitely fast CPU with no latency would start the instrumentation ISR with the counter register equal

to zero. Real processors with more usual latency issues will find the counter at some positive non-zero value that indicates how long the system was off doing other things.

Average latency is just the time accumulated into `total_time` (normalized to microseconds) divided by the number of times the ISR ran (iterations).

It's easy to extend the idea to give even more information. Possibly the most important thing we can know about our interrupts is the longest latency. Add a few lines of code to compare for and log the max time.

Frequent correspondent Dean TerHaar goes much further. He creates a global structure of variables into which each ISR logs start and stop times by reading the timer's counter, generating ISR execution times in addition to latency figures.

Is the method perfect? Of course not. The data is somewhat statistical,

so it can miss single-point outlying events. Very speedy processors may run so much faster than the timer tick rate that they always log latencies of zero (though this may indicate that, for all practical purposes, latencies are short enough to not be significant).

The point is that knowledge is power; once we understand the magnitude of latency, reasons for missed interrupts become glaringly apparent.

Try running these experiments on purchased software components. One package I tested yielded latencies in the tens of milliseconds! **esp**

---

*Jack G. Ganssle is a lecturer and consultant on embedded development issues. He conducts seminars on embedded systems and helps companies with their embedded challenges. He founded two companies specializing in embedded systems. Contact him at [jack@ganssle.com](mailto:jack@ganssle.com).*