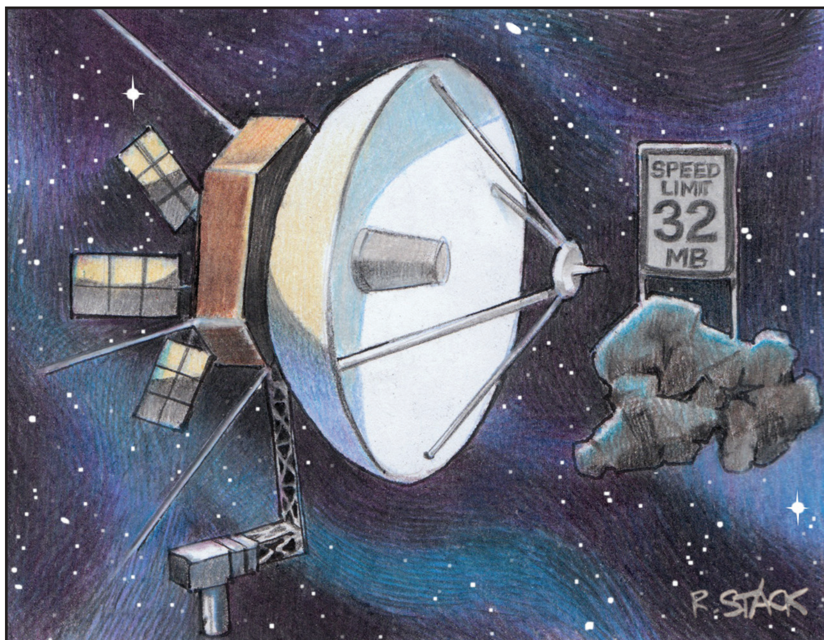
Editor: Gerard J. Holzmann
NASA/JPL
gholzmann@acm.org

Out of Bounds

Gerard J. Holzmann

IN SOFTWARE CERTIFICATION courses at JPL we emphasize that writing reliable code requires the development of a keen sensitivity to system bounds. In any computer only a finite amount of memory is available to perform computations, only a finite amount of time is available to do so, and every object we store and modify is necessarily finite. All resources are bounded. Stacks are bounded, queues are bounded, file system capacity is bounded, and, yes, even numbers are bounded. This makes the world of computer science very different from the world of mathematics, but too few people take this into account when they write code. Perhaps the problem is that in most cases it doesn't matter much if you're aware of the limitations of your computer because things will work correctly anyway, most of the time.

There's a great *Peanuts* strip in which Peppermint Patty explains to a schoolmate that algebra isn't really all that hard: "x is almost always eleven, and y is almost always nine," she explains patiently. In math, if x and y are both positive, then it's clear that their sum must be larger than both x and y. Not in a computer. If we use signed 32-bit integers, this property holds only if the sum is less than about two billion. Because that will be true most of the time, it's safe to say that when we increment x with a positive value, the result will "almost always" be greater than the original.



Almost Always

Assume we're maintaining a signed 32-bit counter. We start it off at zero and increment it once every second. If we keep doing that, the counter will take about 68 years to overflow. This is in fact how the number of seconds was stored in the original Unix operating system, with a clock value of zero corresponding, by convention, to 1 January 1970. This method for recording time clearly has limitations. The 32-bit Unix clock can record only a time span of about 136 years: from December 1901 (recorded as negative numbers counting seconds before January 1970) until January 2038. If we keep a Unix box running long enough, the equivalent of the Y2K problem will occur on Tuesday, 19 January 2038, when time will appear to switch back to 13 December 1901.

Fixing the Unix clock isn't hard. If we simply move the counter to a 64-bit number, the overflow won't occur for about 293 billion years. Even though this amount of time isn't unbounded, it's likely long enough, given that our solar system is expected to come to a fiery end before this counter can reach a mere five billion years.

Things change if we start counting time at a finer resolution than seconds. If, for instance, we increment our 32-bit counter once every 100 ms, a signed counter will overflow in about 6.8 years, an unsigned one in 13.6 years. If we increase the precision to one increment every 10 ms, a signed counter will overflow in 248.6 days, an unsigned one in 497.1

 0740-7459/15/$31.00 © 2015 IEEE

days. Remember those numbers; we'll see them again. Table 1 summarizes them.

Spacecraft Time

The Deep Impact mission launched in January 2005. Its embedded controller calculated time as the number of 100-ms intervals that had elapsed since 1 January 2000. Adding 13.6 years gives us a date in August 2013, well beyond the originally intended lifetime for this mission. The spacecraft was designed to launch an impactor to collide with the comet Tempel 1 and study its composition from the resulting dust cloud. The spacecraft completed that mission on 4 July 2005.

Spacecraft often pick up additional duties after successfully completing their primary mission, provided they have sufficient resources left. On a first extended mission, Deep Impact completed a flyby of the comet Hartley 2 in November 2010. After that, it still had enough fuel for another extended mission, a flyby of asteroid 2002 GT. If all had gone well, the now-renamed spacecraft EPOXI would have reached that asteroid in 2020.

But it was not to be. The clock counter on the spacecraft overflowed on 13 August 2013, which triggered an exception. The exception tripped a reset of the spacecraft that should have put it into its safe mode. Naturally, the reset cleared everything in memory except the spacecraft clock, which meant that the spacecraft ended up in a reset cycle. Even in this desperate mode, ground controllers can do things to recover a spacecraft by issuing "hardware commands" that bypass the main CPU. However, this type of intervention must be done quickly, before the spacecraft loses track of earth and can no longer receive commands. The help didn't come in time, and the spacecraft was declared lost on 20 September 2013—killed by a 32-bit counter.

TABLE 1 How time resolution affects counter overflow.

If you increment a 32-bit counter once every	A signed counter will overflow in	An unsigned counter will overflow in
1 s	68.1 yrs.	136.2 yrs.
100 ms	6.8 yrs.	13.6 yrs.
10 ms	248.6 days = 0.68 yrs.	497.1 days = 1.36 yrs.

It is, alas, not the only example of integer overflow causing problems even in safety-critical systems.

Airplanes

A Boeing 787 passenger plane (the Dreamliner) has many parts, many of which have embedded controllers. Those controllers are likely to have internal clocks and count time. The embedded controllers in the Boeing's GCUs (generator control units) maintain time in 10-ms increments. The corresponding counters are stored as signed 32-bit numbers. At this point, you might want to glance back at Table 1 before reading on.

On 9 July 2015, the US Federal Aviation Administration issued an airworthiness directive (AD) to all airlines instructing them to reboot the GCUs on all Boeing 787s at least once every 248 days.[1] The directive said, "We are issuing this AD to prevent loss of all AC electrical power, which could result in loss of control of the airplane." It explained, "This AD was prompted by the determination that a Model 787 airplane that has been powered continuously for 248 days can lose all alternating current (AC) electrical power due to the generator control units (GCUs) simultaneously going into failsafe mode. This condition is caused by a software counter internal to the GCUs … that will overflow after 248 days of continuous power."

We already know where the 248 days came from, but it's still remarkable to see how often this type of programming flaw can occur in practice, even in systems that have passed fairly rigorous certification. The software for a commercial airplane is generally developed and tested with great care. The development process must comply with the stipulations of the DO-178B standard (and its successor DO-178C). The Boeing 787 software was certainly not written in a rush. The first 787 was shown publicly on 8 July 2007, and the first flight took place on 15 December 2009. The first commercial flight was two years later. We can assume that the plane, and its control software, had been in development for a good number of years at that point. When the overflow problem was discovered in early 2015, close to 300 of the planes were in operation.

Resource Limits

It seems counterintuitive that failures in highly complex systems can have these embarrassingly simple causes. The reason might be that the really difficult design issues typically get ample attention; it's the simple stuff that can get neglected. This can explain why hitting a known resource bound can bring a system to the brink of failure. Examples are easy to find; here I describe two that were in the news fairly recently.

LightSail

On 20 May 2015, the Planetary Society launched a small spacecraft for a test flight. The mission, called LightSail, aimed to test a large solar sail. The test flight hit a snag in just two days. A mission update from 26 May described the problem: "LightSail is likely now frozen, not unlike the way a desktop computer suddenly stops responding."[2]

The cause was relatively simple. An onboard log file stored a record of all telemetry data transmitted by the craft; when that file exceeded 32 Mbytes, it crashed the flight system. Why 32 Mbytes? This wasn't explained in the press releases, but this limit matches a limit of the old FAT12 file system, which some embedded systems still use. The FAT12 design used 16-bit addresses to store sequences of 512-byte blocks in the file system, and yes, $2^{16} \times 512$ bytes comes out to 32 Mbytes.

Curiosity

Another example is much closer to home for me. The Mars Science Laboratory was designed and built at the lab where I work. It successfully landed Curiosity, a large rover, on the surface of Mars on 6 August 2012. Since then, the rover has been functioning reliably, but its software has experienced a few glitches along the way.

One of those glitches occurred almost six months after the landing, on 27 February 2013. The trigger was the sudden failure of a bank of flash memory. The rover software uses flash memory to maintain a file system for storing temporary data files, containing telemetry and images from the mission, before they're downlinked to earth. The software was designed to place the flash file system in read-only mode when something unexpected happens. So, when the bank failed, that's precisely what happened.

During normal operation of the rover, many tasks use the file system to store temporary data products. Because the flash hardware can be slow and a large amount of data must be stored, the software uses a buffering method that can temporarily store data in RAM before it's migrated to the flash file system and then downlinked to earth. All this worked perfectly, almost always.

Knowing that all resources in an embedded system are bounded, we can ask, what happens when the RAM fills up? If this system is designed properly this should happen rarely, and it shouldn't last long because the data temporarily stored in RAM will eventually drain to flash memory. So, in this rare case, the rover computer suspends the data-producing tasks until enough RAM frees up for them to resume executing.

We can also ask, what happens when the flash memory fills up? This event should be even rarer because mission managers are required to keep a substantial margin of flash memory free throughout the mission. However, if it does happen, the software is designed to start freeing flash memory by deleting low-priority files until a sufficient margin is restored. The higher-priority files that live in flash memory eventually will be transmitted to earth, and the space they occupy can be freed again.

All of that looked solid enough to pass all design reviews, code reviews, and unit tests that try to push the known resource limits. But one case wasn't tested—the one that happened on Mars in February 2013. When the flash memory is in read-only mode owing to a hardware error, no further changes can be made: it can't add or delete any more data. The RAM now slowly fills up with new data products, waiting for those data products to migrate to flash memory. At some point the RAM can't accommodate any more data. The data-producing tasks are now suspended one by one, until all activity on the rover freezes.

If this story has an upside, it's that even in this seemingly desperate case, the ground controllers recovered the spacecraft and restored normal operations within days. And our log of lessons learned grew by one more item. We can only hope that the log will be bounded, too.

Should you be worried? Programmers do, of course, know that all the resources they work with are bounded. But it can be hard to keep reminding ourselves of this sobering fact. Given the frequency with which bounds-related problems happen, it's wise to plan for them. Perform deliberate unit and system tests that reach, and try to exceed, known system limits. When using a 32-bit clock, perform tests with the clock set forward 248 days, 1.3 years, and 13.6 years, and see whether the system still works correctly. Of course you'll have a long time to think about it if you forget one of these tests, but you'll surely sleep better if you don't.

References

1. "Airworthiness Directives; the Boeing Company Airplanes," *Federal Register*, vol. 80, no. 84, 2015; http://rgl.faa.gov/Regulatory_and_Guidance_Library/rgad.nsf/0/584c7ee3b270fa3086257e38004d0f3e/$FILE/2015-09-07.pdf.
2. M. Wall, "LightSail Solar Sail Test Flight Stalled by Software Glitch," Space.com, 27 May 2015; www.space.com/29502-lightsail-solar-sail-software-glitch.html.

GERARD J. HOLZMANN works at the Jet Propulsion Laboratory on developing stronger methods for software analysis, code review, and testing. Contact him at gholzmann@acm.org.