The Next Date Crisis and the Ones After That

Robert L. Glass

Reality is the murder of a beautiful theory by a gang of ugly facts.

ROBERT NEUBECKER

I'm sure you already know about how all information systems will go berserk when the year 2000 comes, the so-called "date crisis" everyone is talking about.

This column is about a date crisis. But not *that* date crisis. I want to talk about the one that comes *after* the one everyone knows about.

First, to make sure we're all on the same page, let me describe the date crisis everyone already knows about. Dates in most information systems are stored in the form MM/DD/YY. That is, two digits are used for each month, day, and year. That's not a problem for months or days, of course.

But years? Big problem. Even though most of us say "the 90s" and in many other ways use years as if two digits were enough, we're approaching 2000, when two digits simply aren't sufficient to do the job. Lots of information systems do arithmetic on dates (e.g., how old is a person whose birthdate is 02/03/32?). Information-system arithmetic will fail come the year 2000. (By the way, here's the answer to the question—in 1999, 67. In 2000—"minus" 32.)

Remember, this column is *not* about that date crisis. I'm assuming nearly everyone has read and listened to some of the thousands of words spilled over the last several years on this topic. And I'm also assuming nearly everyone is aware the crisis is almost inevitable because hardly any companies have allocated and spent the resources necessary to fix the problem. There *will* be a crisis at the turn of the century, although it probably won't be the World War III some consultants are hyping it to be.

What, then, is the date crisis I want to discuss?

No matter how—or if—an enterprise solves its year-2000 date crisis, another one is coming. And the form of the solution chosen for this particular crisis may determine when the next crisis will occur.

First of all, let's state the obvious: There is no permanent solution to the date crisis. Whereas the months of the year recycle after 12, and days of the months recycle after about 30, years just keep marching on toward infinity. And computers with their finite word lengths cannot hold them forever. That explains part, but not all of, the next date crisis.

Let's say you decide to employ what to most people in 1997 is the obvious solution to this date crisis—the four-digit year. You change all your data files and data declarations to accommodate a couple more digits per year, test all the software affected by the change, and sit back and sigh the sigh of satisfied relief: one more crisis solved. Not so fast. Your solution is certainly valid for the foreseeable future, of course. But it will fail come the year 10000. As surely—and in the same way—as the 2000 date crisis.

It's hard to get excited about a future 8,000 years ahead. For one thing, you're probably saying, "What are the odds that information systems as such will be around that long?" Except that's the same kind of thinking that got us into the year 2000 mess. No one believed, back in 1975, the information systems of that day would last another 25 years.

Still, no one is *really* going to worry about the four-digit solution. After all, the five-digit solution—or any other solution you contemplate—will also have its very own day of doom associated with it. How far off is it necessary to postpone this problem?

There are other, more sinister, forms the next date crisis can take. In fact, the Unix operating system apparently has its own built-in date crisis ticking away like a time bomb, and it's difficult to predict precisely when this particular bomb is going to explode.

But for now let's continue to talk about IS applications. Most enterprises, as I've said, will employ the four-digit-year approach. Some are troubled by the expansion of databases if all the two-digit years get converted to a bigger number.

Because of that, some companies are trying another solution: Keeping the two-digit dates on file, but having a procedure in the program to convert each date before its four-digit equivalent is used. So far, so good. This solution contains a serious problem, however. For instance, how does the procedure decide whether the date 56 represents 1956 or 2056? The answer is usually a bad one, going like this: Pick an arbitrary date that will be accurate today and use it as a watershed date. For example, any year less than 50 is in the next century, and any greater than or equal to 50 is in the previous one. Ergo, 56 = 1956. Problem solved?

Today, perhaps. But what happens as we approach 2050? The solution comes unglued, just as badly as the year-2000 problem. Or perhaps worse, because the solution is local to the enterprise (other companies might have chosen 60 or 70 instead of 50) and thus there will be fewer cries of warning next time. But no one would use this solution, you may say. Not true. A consultant writing in a leading journal proposed it within the last 10 months. How many companies are following that very bad advice?

The point is the date crisis is a "pay me now, *and* pay me later" kind of problem. There *will* be another date crisis sometime in the future, and the decisions enterprise IS people make today will determine when that next crisis will be.

At this point, you may find yourself sympathetic with those concerned about the increase in storage consumed by the four-or-more digit year. Given that the "greater than 50" solution is a bad one, is there any other solution that will prevent database expansion?

Turns out there is one and hardly anyone is talking about it. Because dates have always been kept in decimal form in our binary (and, before that, decimal) computers, we assume blindly it must always be so. In doing so we neglect another form of data that computers use—binary. Suppose we store the year in binary rather than decimal form?

Most years today, being in YY form, occupy 16 bits (two eight-bit bytes). (This is not universally true. For example, Cobol has a numeric form called "packed decimal" that alternatively allows digits to be stored in four-bit form, so with packed decimal, a year would require 8 bits.) If a number is stored in binary form in 16 bits, it can be as large as 65,536 (64K). That is, using the same two bytes that YY occupies, a year of up to 65536 could be stored without increasing the memory size in the database. In order to use this solution, however, the database would have to be rewritten with years in binary form, the program logic would have to declare years as binary (there's a form of *computational* in Cobol that will do the trick), and all the program date logic would need to be checked to make sure nothing more is affected by the change (it is possible the logic may not need to change, except for human-readable outputs). (Note this solution is not valid for packed decimal dates, discussed earlier. Whereas 16 bits gives 64K, 8 bits only gives 256, insufficient to store years.)

So in this "pay me now *and* pay me later" game, you pay your money and you take your chances. You can have another date crisis in the next

NO ONE BELIEVED, BACK IN 1975, the information systems of that day would last another 25 years.

century (around 2050 or so), at 10000, or at 65536. The choice is yours. There are pros and cons to each approach. Only the technologist in charge of an enterprise's suite of applications can make the best decision, because there is no universal best one. Although the four-digit solution comes close.

And what about the Unix-unique date crisis mentioned earlier?

Here's my understanding of what happens in Unix. Bear in mind that I am *not* a Unix guru. It may very well be that I'm wrong. I hope I am. But if I am correct there may be a date crisis for applications on Unix systems that could become the mother of all date crises.

Unix has a device called "Unix time," the measurement of time in seconds from a base time—the first day of 1970. Unix time, by 1997, has gotten quite large. There are roughly 31 million seconds in a year. In 27 years, we have piled up 27*31 million, or 837 million seconds. The number is constantly increasing—by, for example, 300 or more since you started reading this column.

Numbers in computer speak are finite. No matter how large a fixed space you set aside, as numbers charge madly toward infinity they will outgrow their space. Will Unix time outgrow its space? Inevitably. The question is, when?

The answer is hardware dependent. A computer's word length probably determines the date on which Unix time will overflow. (Go from the biggest number back to 0 when the capacity is exceeded). Let's make a typical case assumption and say a computer has a 32-bit word. If my arithmetic is correct, Unix time will overflow around 2035. Sooner than even the worst of the date crisis solutions contemplated, and even sooner for smaller-word processors.

This is one of those "surely I'm wrong" kinds of findings. Surely the designers of Unix anticipated such a problem and have provided for it. Does anyone in my reading audience know? If so, tell me and I'll pass the answer on in a later column. In a way, it sounds like I'm crying "fire" in a crowded theater. If there is no fire, it's important I be set straight. But if I'm right, it's time to begin talking about what Unix and its application programmers must do.

So we see that there's yet another software date crisis (YASDC) coming. That if you're using Unix you may have very little choice about the date it arrives. That for all IS applications, using Unix or not, there's no choice as to *whether* there will be YASDC. Your only choice is *when* to schedule it. ◻

Robert Glass *is the publisher of the* Software Practitioner *newsletter and editor of Elsevier's* Journal of Systems and Software. *He welcomes feedback: 1416 Sare Rd., Bloomington, IN 47401.*