Double Zero

July 1997 / Features / Double Zero

Year 2000: no COBOL, no mainframe, no problem, right? Wrong. Place your bet correctly on the year 2000 roulette wheel.

Joe Celko and Jackie Celko

It's the end of the world as we know it. At least, as the year 2000 approaches, many people are warning of a computer apocalypse. Some of these doomsayers are software vendors and consultants who are trying to sell salvation for you and your computer system. There are several questions you will want the answer to. Should you buy? How big is the year 2000 problem, anyway? Whom will it affect? Will PC users be immune?

Well, the year 2000 problem is real , and the ramifications will be huge. No one will be co mpletely immune, although some firms and individuals will experience fewer problems than others. Some industry experts are esti mating that the worldwide cost of fixing -- or surviving -- the year 2000 problem will be between $400 and $600 billion.

The year 2000 problem will affect every government agency, business, or individual that uses a computer. Even if you do *not* use one, you will still feel the impact when you interact with anyone who does. If you have insurance, buy airline tickets, or want season passes to a sporting event, you may have trouble. The automated systems that control your building's heating and cooling and your cherished ATM may refuse to work.

Businesses that do not prepare properly for the year 2000 may not be able to survive it. For starters, businesses whose computer systems are not year 2000-compliant may be unable to obtain business insurance. Mistakes in calculating interest or a delay in payments, automatic checks, or premium n otices may expose a company to legal risk. And nothing motivates like legal exposure.

A company may lose revenue because its computer will not recognize a purchase order with a date after 1999. A company may refuse to order from your company or refuse to enter into a joint venture with your company because your computer system cannot handle year 2000 dates in a standard way.

Even without the year 2000 problem, date-format conventions are confusing. The date "December sixteenth of the year one-thousand nine-hundred and ninety-seven" is done in many different ways. You would find it as 12/16/97 in Boston, 16/12/97 in London, 16.12.97 in Berlin, and 97-12-16 in Stockholm. Then there are conventions within industries within countries. For example, the U.S. military would write that date as 1997-Dec-16.

Software packages typically have a general way of formatting dates for display. The usual tools allow a mixture of a two- or four-digit year, a three-letter or two-digit month, and a two-digit day in the month. Slashes, dashes, or spaces separate the three fields.

ISO Dates

However, there is only *one* real international standard: ISO 8601:1988 "Data elements and interchange formats -- Information interchange -- Representation of dates and times." It specifies the all-numeric yyyy-mm-dd format.

The National Institute of Standards and Technology (NIST) has approved the use of slashes instead of dashes in the U.S., keeping with the older U.S. convention. (It is also interesting to note that some vendors, notably Microsoft, claim to be trying to achieve year 2000 compliance, but sell products that cannot display in the ISO 8601 date format at all.)

You can divide your potential year 2000 problems into two familiar categories: hardware and software. Of the two, software will be the most problematic.

Most of the hardware problems that happen to you will have a single universal solution. Unfortunately, however, the various combinations of packages, languages, and in-hou se applications that are all interacting in a single installation will make it necessary for companies to develop a unique solution for each software system.

The Odometer Problem

Hardware problems occur when a system will not accept years greater than 1999. We call this the odometer problem because it is in the hardware and not in the application code. This is not the same as the millennium problem, where date representations and arithmetic are invalid. Think of the odometer in a car that has reached its upper limit and turns over to all zeros.

One example is a manufacturer of specialized fiberglass cloth who uses custom-built looms that are more than 40 years old. Each bolt of cloth is individually time-stamped by the looms. The time stamp uses a two-digit year code, and its control is by hard-wired circuit boards with transistors. This is clearly an extreme example of a hardware problem.

The odometer problem exists in both mainframes and PCs. For example, the Unisys 2200 system wo uld have failed on the first day of 1996 because the eighth bit of the year field -- a signed integer -- went to one. The vendor was able to solve this problem.

Other internal date representations have different failure dates, but many of them fall in the first century of the next millennium. Mainframe vendors are working on solutions that will let their hardware continue to function into the twenty-first century. However, users with very old equipment -- such as the manufacturer with those looms -- may be on their own if the vendor refuses or is unable to support the equipment.

Intel-based PCs also have odometer problems. How the system clock will wrap around depends on your BIOS chip, but the most common dates to which it will reset are 1900, 1980, and 1984. You can test your computer. Set the date and time to 1999-12-31 (in whatever input format your machine expects) at 23:59:51. Let the machine run 10 seconds so that the clock rolls over. What happens next depends on your BIOS chip and DOS ver sion.

The result may be a date display that shows 01/01/00, so you think you have no problems. Although the *display* may appear to be correct, the clock may read 1980-01-01 or 1900-01-01 internally. You may find newly created files with dates in the twentieth century, because the OS accessed the clock directly and fetched the incorrect date.

This problem passes along to applications, but not always in the way you would think. Quicken 3 for the PC running on DOS 6 is one example. As you expect, directly inputting the date 2000-01-01 results in the year resetting to 1980 or 1984 off the system clock. Strangely enough, however, letting it wrap from 1999-12-31 into the year 2000, Quicken interpreted the change as 1901-01-01 and not as 1900. This indicates that Quicken sometimes references the internal clock and sometimes uses a date the user inputs. The date referenced from the clock and the date input manually into the system are both incorrect and different.

A number of widely available freeware programs do year 2000 tests on PCs. These include DOSCHK from Bob Stammers, 2000Test and 2000Fix from Dan Goodell, and Year2000 from Tom Becker (Air System Technologies).

Other PC applications known to exhibit year 2000 difficulties include Microsoft Access, FoxPro, and Visual Basic; CA Clipper; Borland Delphi; Gupta SQLbase; and Oracle. Fixes or workarounds for many of them are widely available as freeware.

Because the year 2000 is two-and-a-half years away and the life span of a PC in some environments is shorter, some managers believe the problem is self-limiting. There are, however, many older machines out there. What usually happens to old machines in a corporate environment is that they stay in service but pass down the line to a less-critical function as newer models replace them.

Another solution is to replace the chip. This is expensive and time-consuming when you have a large number of machines. A software patch may not be enough if a program reads the clock for itself or if the patch does not work with existing programs.

Some vendors are ensuring that machines made after a certain date are problem-free. These vendors include AST Computer, for machines that were made after July 1996.

Some generic Pentium-processor machines have their BIOS in a flash EPROM that you can directly upgrade. If you have a BIOS chip from Award or AMI manufactured later than October 1995, you may be able to upgrade it using a patch from the manufacturer's Web site.

Some earlier machines are also upgradable, so check your documentation. Many other manufacturers are supplying or will soon supply user-upgradable systems. Again, check the documentation. No manufacturer will guarantee that the patches will fix all hardware-related date problems, but as the year 2000 approaches, manufacturers will continue to offer improved patches.

Many motherboards allow user upgrades, but sometimes you must perform the upgrade in the proper sequence. Some motherboards have a jumper switch that you must set before upgrading the system. If you do not follow the proper procedure, the system will return an error message. Even worse, the upgrade will appear to have worked but will actually have done nothing.

Planning Your Campaign

David Eddy of Global Software states, "While about 99 percent of what's been publicly written about Y2K so far is essentially 'Y2K is a COBOL mainframe problem,' this absolutely is *not* the case. Software is software. To the best of my knowledge, there is no language (even the Department of Defense's Ada) that forces the features and capabilities of that language to actually be used correctly."

Software problems caused by the year 2000 will be widespread, complex, and costly to repair. According to Capers Jones of the Software Productivity Consortium, the cost to get your software ready for the year 2000 will vary by language and industry. For example, programs written in COBOL will cost about $28 per function point. Programs written in assembly languag e will cost more than $75 per function point, and programs written in object-oriented languages will cost less than $18 per function point.

Old legacy systems written in-house will be the most costly to repair. In many cases, the source code and even the specifications may no longer exist. Upgrades and problem fixes may have no documentation. Old packages may no longer have support, even if the original vendor is still in business.

Some new packages that claim to be year 2000-ready will still fail under the right combination of factors. That includes software written for PCs. Spreadsheets are the worst offenders, depending on how internal date-handling functions work, but any package that uses a calendar or scheduling function is at risk, including databases. When these packages feed data to a network computer, the contaminated data passes along to your entire system.

Even bringing your own system into perfect compliance may not be enough if you interact with other systems and bring contamin ated data back into your own system. Lack of standardization of date formats coupled with what year 2000 compliance entails will continue to be a problem.

Your Options

There are four methods for dealing with the year 2000:

1. Do nothing. Wait to see what blows up and fix it when it does. This is the choice of a surprising number of small- and medium-size firms. This ostrich approach may be attractive -- especially if you plan to retire by the year 2000 -- but remember: You will be counting on the company computer to do your retirement benefits correctly.

2. Replace everything. Small firms can frequently afford to replace their software because they do not have large historical databases. If they choose not to replace their application programs, they can frequently upgrade them and get a fix from the vendor.

3. Do simple fixes. Many products offer a simple fix for existing software. You may not have to change the data or may have to make only small changes.

4. Do a full a nalysis and complete fix. This involves testing and analyzing the system for potential date problems and making changes to the system so that it will handle the year 2000 correctly. The system must change internally to handle a four-digit year and change at the point of input so users must input a four-digit year.

The simple fix usually involves a *pivot point* (see the figure "Effects of Pivots on Two-Digit Years"). This approach, which is also called a time frame or epoch setting, lets you patch applications without changing the data by making an assumption about the century in which a two-digit year falls. Using a particular year as a pivot point, the system adds either 1900 or 2000 to the two-digit year to make it a four-digit year. For example, if the program is looking at PC software, it's safe to assume that 70 and above belongs in the 1900s, so 70 would be the pivot point. If the program is looking at kindergarten-age children, a pivot point of 90 might be appropriate (although you might turn up some centenarians in the process).

This is not a new idea. Look at printed forms such as personal checks. Many forms have the date space preprinted as _____19__, under the assumption that the century is the twentieth.

Many packages use a pivot point in a variety of price ranges for the most popular languages and platforms. There are advantages to using a package that relies on a pivot point to handle your year 2000 problem. It is less expensive than a full analysis and repair of the system, and it requires less user training and involvement. In many cases, the end user continues to key two-digit dates and all the work happens behind the scenes.

A disadvantage of this method is that you must pick a pivot point between 00 and 99, and there is no reason to favor one pivot point over any other. If Microsoft's lead becomes a de facto standard, the two-digit years between 00 and 29 will convert to 2000 to 2029, while 30 to 99 will convert to 1930 to 1999. This makes the birthday of a person born in 1920 wrong. Quicken maps the two-digit years 00 to 50 into the years 2000 to 2050. This makes calculating the return on a stock you bought in 1949 wrong. In short, there may be no perfect universal pivot point for a given application.

Within this class of solutions, you can store the dates as either four digits or two digits. Storing them as four digits means you are using the two-digit year format as a convenience for data-entry clerks or users. If you do this, you should display the full date in the proper format for all reports.

If you keep the year as two digits internally, you will have problems. When two of your own applications disagree on the pivot point, in effect, every date has its own user-defined data type. You will need to write conversion routines to use both types of dates in the same calculations. If you do not have control over both programs because one of them is external to your system, you risk internal calculation errors that will be difficult to detect. If only a tiny percentage of the dates used to calculate interest rates are incorrect, the incorrect dates will be difficult to identify -- but the calculation errors could have an enormous impact.

Only a small part of your system will lend itself to this type of fix. This is a quick-and-useful patch, but it's only for systems with short time horizons or that are due for replacement soon.

Big System, Big Changes

You must analyze medium to large systems as a whole and make repairs to bring them into year 2000 compliance. For many firms, the year 2000 effort will be their largest data-processing project ever.

When Data Integrity ran a test of 1,000,000 lines of COBOL produced by a variety of industries, it

discovered that less than 13 percent of the COBOL modules were affected and that less than 0.5 percent of the total code had to be changed (see the figure "Finding the Needles in the Haystack"). This test indicates that a good front-end analysis to locate problem areas and assess their impact is essential. COBOL-oriented analysis products include Adapt/2000, from Allegiant Legacy Solutions.

The analysis phase of the project may take up to a year. A lengthy up-front analysis clearly can save you time and money over the life of the project. Anne White, marketing manager of Isogon, believes that a big part of the project is to identify and eliminate dead code. Frequently, 20 percent to 30 percent of the code in large mainframe systems is no longer used.

An analysis of a Texas oil company indicated that 40 percent of its code was dead and could be deleted. Isogon calculates the cost of bringing a system into year 2000 compliance to be about $1.50 per line of code. By eliminating 40 percent of its code, the company saved 40 percent of the conversion cost.

As you go through the steps to identify dead code, products such as Isogon's Audit 2000 will also help you identify your most mission-critical, active code, so you can begin working on the necessary changes. Graham Thompson of Global Software stresses that you cannot analyze one program in isolation from the system. The program that collects the data may be quite distant from the program that uses the data. It is important to look at JCL and utility programs for sorting parameters and follow the data through the system as well as within individual programs.

Fifty percent of your project's budget may be for testing. Some shops have adopted the approach of using the initial test data to retest the system, but much of the test data and many scenarios were developed before the year 2000 was a serious consideration. QES/EZ, from QES Software, lets end users develop models and use them to identify problem areas caused by the year 2000. After you fix the problem, the same model can retest the system. The system does not need to be shut down during testing. The product is PC-based and operates independent of language or OS.

Weird Dates

It is important to begin your year 2000 proje ct as soon as possible. In a large firm, the project may take longer than a year. While some experts are saying that you must complete the project by June 1, 1999 -- the start of the 1999-2000 fiscal year -- in reality, for many firms, the actual date is January 1, 1999. This is because many programs, particularly those developed in-house, use the number 99 to indicate indefinite or unknown information. That's only one example of a "weird date."

Some common problems that you can expect to encounter as you analyze your system will be the result of such weird dates. Weird dates are a result of programming languages not having a date data type built into them. This forced applications developers to write their own date routines. Besides letting developers use the year-in-the-century format and create all the problems associated with it, it also let them permit temporal data values that are not really dates in these fields.

There are two basic species of weird dates, with many subspecies. For lack of better names, I'll call them dates that are not dates and nondates that are really dates until we can get better names.

Dates that are not dates: The most common examples are *eternity codes* , which come in two basic flavors in old COBOL systems using character fields. The indefinite future was shown as 9999-99-99 or 99-99-99. It meant that the event had not yet happened or might never happen. Likewise, the indefinite past was shown as 0000-00-00 or 00-00-00. It meant that the event had already happened, but we did not know the exact date. For example, you do not know an employee's retirement date, so you use the indefinite future code. You do not know an employee's birth date, so you use the indefinite past code.

The major advantage of these encoding schemes is that these values will sort together either before or after all valid date representations. They also save space in the files, which was a consideration in the old days.

This method was never standardized, and the encoding schem e tended to grow as more codes were added. For example, one state prison system used all 9s and all 8s for life sentence and death sentence, respectively, in the "expected date of release" field that's found in inmate records. Commercial users also found that expiration dates for lifetime warranties could be encoded this way and then broken down into types of warranty -- parts and labor, parts only, labor only, and so forth.

Automatic conversion tools that are trying to move the legacy data to SQL databases cannot handle this type of encoding well. They either put the records in a rejection file or convert all the special codes to NULLs. The proper way to handle this would involve creating a second column in the table to hold a separate code for the reason the date is missing.

Not all eternity codes are deliberate. When you build a data warehouse and have to scrub your data, you will find that data-entry clerks, who ran into situations where they needed an indefinite past or future value in a colu mn, invented their own codes. If the field edits were not rigorous, virtually anything could get into the database.

An interesting example found in legacy data is 1111-11-11, which is a valid date that you can key into a form screen or punch-card field by holding the 1 key down and letting it repeat. Blanks and repeated letters (xxxx-xx-xx) were also another popular option when field format editing was poor.

Nondates that are really dates: The other subspecies of weird dates includes fields that are not dates per se, but that derive from or contain dates. Some common examples are account and serial numbers that begin with the year in the century. For example, the first account number issued by such an application in 1997 might be 97-0001, the second would be 97-0002, and so forth. This is a common numbering pattern in the insurance industry.

As in the case of the first subspecies of weird dates, multiple facts have overloaded the field, and the data is not normalized. Programmers who knew th e numbering pattern wrote code based on the implicit ordering between the serial number and the sequence in which they occurred. If the pattern continues and the first account created in the year 2000 is 00-0001, all those programs are going to fail.

The most common and dangerous example of this problem is the IBM magnetic-tape label convention. For decades, tape librarian systems automatically assigned a label to the reels of magnetic tape made up of the year in the century and the day in the year (001 through 366). This label number determines which "expired" tapes to recycle, destroy, or erase. Note that tape label 00-001 is lower than tape label 99-365 and therefore subject to earlier destruction.

More subtle forms of this category of problems exist in algorithms that use parts of the system clock or other dates as parameters to produce a result that is not so obviously date-related.

Leap Year

You might remember being told in grade school that there are 365.25 days per year and that the accumulation of the quarter day creates a leap year every four years. There are really 365.2422 days per year, and every 400 years, the fraction of the fractional day left over from the "regular" leap years accumulates, too.

Yes, as if matters aren't complicated enough, 2000 is a leap year. Since most people are not over 400 years old, we have not seen this rule applied until now. Certainly, no computers have. The correct test for leap years in Pascal is the following:

```
FUNCTION isleapyear (year:
INTEGER): BOOLEAN;
BEGIN
IF ((year MOD 400) = 0)
THEN isleapyear := TRUE
ELSE IF ((year MOD 100) = 0)
   THEN isleapyear := FALSE
   ELSE IF ((year MOD 4) = 0)
     THEN isleapyear := TRUE
     ELSE isleapyear := FALSE;
END;
```

Many programs do not use this algorithm. They implement only incomplete versions of it. One such fiasco was the failure of Wacovia Bank to post automatic payroll deposits on 1996-02-29, because its program did not even get as far as doing t he "division by four" test.

The problem is not just in legacy applications that are written in-house. Many packaged programs are also flawed. The date functions in spreadsheets seem to be the worst offenders, but any program with a date function in it may be wrong. Just key in the date 2000-02-29, do some calculations with date arithmetic,

and see what happens.

The year 2000 is coming in less than three years. If you have not started your testing-and-conversion project, the time is now. Many large mainframe vendors such as IBM have year 2000 initiatives and are working to help their customers prepare. The number of consultants working in the area is increasing, and the number of products being offered by vendors is spiraling upward. Time is the resource that is most lacking.

The year 2000 problem originated, of course, with hardware and software designers who did not foresee the turn of the century and plan for it. This lack of foresight became immortal in computers and programs that may go haywire when the calendar flips over from 1999 to 2000. Will today's designers learn from this experience? See you in 2100.

The Process

```
The general steps necessary for a year 2000 project are:

1. Allocate adequate resources. The project will require a full-time
   manager. Be aware that the preparation-and-testing phase will take
   months.

2. Choose products and consultants that
do not
 offer a
   one-size-fits-all approach. You will probably need several tools.

3. Analyze and test hardware, programs, and database files.

4. Eliminate dead and redundant code.

5. Use a simple pivot-point approach on applications that have a
   short time horizon and do not interact with other programs. A library
   checkout system is an example. Pivot-point approaches may also be
   appropriate for systems that are becoming obsolete and that cannot
   justify the expense of a full conversion.

6. Fix or replace remaining problems with date routines.

7
. Retest.
```

Effects of Pivots on Two-Digit Years

illustration_link (22 Kbytes) 

Using a simple pivot point makes some 1900-years correct, but it may turn 1900-years into 2000-years.

Finding the Needles in the Haystack

illustration_link (24 Kbytes) 

Studies show that not much COBOL c ode needs to change -- but it does need to change, and you have to find it first.

Problems with Year 2000 Projects by Company Size

illustration_link (36 Kbytes) 

Midsize companies report the most problems with every aspect of year 2000 projects.

Joe Celko is an Atlanta-based consultant with Northern Lights Software, Ltd. He has been a member of the ANSI X3H2 Database Standards Committee since 1987 and helped write the ANSI / ISO SQL-89 and SQL-92 standards. He is the author of four SQL books. You can reach him at 71062.1056@compuserve.com . Jackie Celko is an Atlanta-based tec hnical writer.