DATE HANDLING BEYOND THE YEAR "99"

by

Kevin Cooper

Hewlett-Packard Company
Pleasanton, California, U.S.A.

Summary

Most computer software that handles dates uses only two digits to represent the year. With the twenty-first century rapidly approaching, we must begin to think about how we will manage dates beyond the year "99". This paper focuses on three issues related to the handling and formatting of dates. What are the best ways to convert software that was written with the two-digit year limitation? What techniques should be used to design and code new programs that will avoid these pitfalls before the year 2000 arrives? And how can future productivity tools make all of this invisible to both software developers and users?

Introduction

Imagine these scenarios:

January, 2000 - You arrive at the office on Monday morning, January 3, ready to begin the new year. One of your users calls with a problem, and you promise to have it investigated immediately. When several more callers have similar problems within the next hour, you begin to wonder if someone has tampered with your computer system. Before noon you realize that you have a serious problem on your hands involving all your programs which handle dates.

June, 1995 - Your company has a business forecasting program which projects out five years. It worked just fine the last time you ran it, which was in December, 1994. But now in 1995 strange things seem to be happening with the program. Figures for the fifth year are showing up in the first column, and all the other years are shifted out one column. What could be causing this?

March, 1986 - I go to the bank to start a long-range savings program for my son's college education. I invest in a long-term note scheduled to mature in 2002, the year he will start college.

The computer printout from the financial institution thanks me for my business, tells me I will receive annual statements of interest earned, and concludes by promising to send me a reminder notice upon maturity - in March, 1902. Something makes me think I should have taken my money elsewhere.

The impending arrival of the year 2000 will force everyone associated with the data processing profession to take a good, hard look at the way computers handle dates. Most software that depends on date-related information today uses a six-digit date with a two-digit year, based on the assumption that the most significant digits of the year will always be "19". The dawning of the twenty-first century will change that sooner than many might think, and the programming staff will not be able to use the excuse that they were not given enough notice! Sort routines will place dates in the year "00" (2000) before dates in the year "99" (1999). And what about the potential confusion when the date 03/02/01 appears on a report? Is this March 2, 2001? February 3, 2001? February 1, 2003?

Before we look further at these problems, let's look back at a little bit of computer history.

Background

How did computers start using six-digit dates in the first place? The normal way people write dates is with six digits separated by slashes. Two people who communicate using the date 03/03/86 both understand this to mean March 3, 1986. (Note that this example avoids the confusion caused because Americans write dates with the month first while Europeans place the day first; we will discuss that later.) When the computer was first being used for data processing, data storage space was at a premium, so the early programmers had no reason to waste two extra bytes of storage space on every date just so they could cover the arrival of 2000. I think they were quite justified in making that choice.

That leaves us in the late twentieth century to solve the problems associated with the six-digit date, or more specifically, the two-digit year. As shown by the earlier examples, this will impact the computer world long before we actually reach the year 2000. It has already affected some organizations, such as financial institutions, who must handle notes and loans due to mature 15 or more years from now. The rest of us will become more aware of its consequences as the time draws nearer.

My first introduction to this whole problem came when I was a novice COBOL programmer working for Hewlett-Packard fresh out of college. I needed to program a six-digit date-editing routine to check whether February 29 was a valid date in a given year, which is really a test for leap years. I am not recommending the technique I chose, but I wrote the code to find leap years after 1977 like this:

```
IF YEAR = 80 OR 84 OR 88 OR 92 OR 96 ...
```

Then I stopped and pondered. What will happen when YEAR becomes "00"? I was appalled as I considered the impact that the turn of the century would have on the application I was writing. Sorts, edits, and countless lines of code which compared two dates would cease to function properly. This first encounter with the problems we will face when the year 2000 arrives made me realize that one of two things must happen before then:

1) I must try to find ways to solve this problem and get the word out so we don't face a computer crisis come January 1, 2000; or

2) I must get out of the data processing field altogether and seek a new career.

This paper represents my attempt at the first alternative. Hopefully, this means we can all avoid the second option.

By the way, one of the interesting sidelights about the year 2000 is determining whether or not it is a leap year. The general rule is that years divisible by four are leap years; based on this rule, 2000 should qualify. However, this rule has an exception. If a year is divisible by 100, then it is <u>not</u> a leap year. That seems to indicate that February, 2000, will only have 28 days, right? Well, no; there is another exception to the first exception. Years divisible by 400 <u>are</u> leap years. I had never heard of this 400-year rule until I saw it coded in someone else's date-editing routine, so I checked my almanac and found that the year 2000 truly is a leap year, with February 29 falling on a Tuesday. This means that leap year tests which ignore the exceptions and just check if the year is divisible by four will work until 2100 arrives. That is long beyond the expected life span of any computer program in existence today.

<u>Conversion Methods</u>

The first issue for discussion is the conversion of software that was designed with the limitation of a two-digit year. This problem may seem like it is a long way off, but we cannot avoid the issue for many more years. We have already seen that programs which project out to future years will have problems well before the twenty-first century arrives. Even if we assume that the majority of date handling programs will only have errors when we actually reach the year 2000, we are left with less than 14 years to completely eradicate the problem.

So what software should we be concerned about converting? To answer this question, we must try to figure out what software will still be around in the year 2000. One possible gauge of that is to look back 14 years and see what software from 1972 is still around today. Two obvious examples in the HP 3000 arena are the MPE operating system and the IMAGE date base management system. While most of their code has been rewritten over this time span, much of the original designs remain, such as the six-digit date used at system startup by MPE. There is no way to know whether these designs will still be around in the year 2000, but we can see from these examples that some of the software being developed today will in all likelihood survive the 14 years left until we reach the year 2000. Unless these new programs are designed and coded using some of the techniques to be discussed later, they will be the ones which require converting as the next century approaches.

Given the high probability that some software will need to be converted, when should all the necessary changes take place? Do we need to start converting our entire collection of existing software right away? No, I don't think so. Based on a typical life span of five to ten years, most of the computer programs in existence today should be obsolete by 2000 (although that is not the same thing as being completely out of use!). The time to evaluate this type of conversion will be a year or two <u>before</u> each software system begins to malfunction. For most software that means the late 1990's, which certainly gives us plenty of time to prepare!

Even so, I predict that the upcoming arrival of a new millennium will catch many of those in the data processing profession by surprise. Conversion will be a big issue because organizations will not realize the magnitude of the effort ahead of them. I would like to present three possible conversion methods to help you estimate the effort that will be required when the time comes.

1) <u>Full Upgrade</u>. This technique requires converting <u>data</u> and <u>programs</u> to use a four-digit year. This is the most direct approach because it is a complete conversion that involves no gimmicks. But it may require a massive effort to find every occurrence of a date field, change it, recompile every program which references it, and test it all thoroughly. The source code for every program must be available, and the current compilers of the day must be able to successfully recompile that code. The amount of time for testing should not be underestimated, even though this seems like such a straightforward approach. Because of the extra data space required, the possibilities for stack overflows and inefficient blocking factors (or whatever errors future computer architectures give us) are endless. Reports may not be able to accommodate two extra columns of data, so conversion routines will probably need to be written to print the

date with a two-digit year. People will have no trouble when they see "00" as the year after "99" - that only confuses computers!

2) Logic Upgrade. This technique leaves data as it is but implements some additional logic to handle dates properly. For example, it may be reasonable to assume that years above "50" belong to the 1900's while the rest belong to the 2000's. This method makes sense for programs which frequently use dates with two-digit years but do few comparisons (such as determining which year is greater). In these cases, the cost of a full upgrade as described above may be too high to justify. These new date comparisons could be implemented as common routines which could be called by many different programs. While this is less costly than a full upgrade, we must still be able to locate and recompile the source code. The effort to test should be less than that required for a full upgrade because fewer changes are being made, but again this should never be underestimated.

3) Data Upgrade. If source code does not exist or cannot be recompiled successfully, the only option besides a total rewrite is to attempt a data conversion of some kind. This can get tricky, but it just may work. If dates are stored in character format (rather than packed or integer), the ASCII collating sequence contains several special characters which have an internal value less than the value of the character "0" (zero). At the time when year "00" is about to be introduced to the system, all the two-digit years beginning with "9" can be updated to start with a special character such as blank or asterisk (*). This will force dates with year "00" to sort after dates with year " 9" or "*9". Users will see years of " 9" or "*9" on their screens and reports whenever they access the old data, but people can be educated in advance about the change. This will work best in instances when the older dates will be aged off the data base soon after the conversion. There is some risk here that the strange-looking dates will fail a logic edit or cause other problems, but this technique can be viewed as an acceptable work-around when the only other choice may be to scrap the programs entirely. A similar approach would use characters like "A0" to represent the year 2000, since they will sort after "99"; but such dates would be harder to work with and would remain in our data for a longer time. As with the other methods, any changes must be thoroughly tested in each environment before they can be considered successful.

New Design Techniques

As we have already stated, the best way to avoid the problems associated with conversion is to design and code new programs today in a way that eliminates the basic problem. It is not too early to begin using this strategy. Most of the software that will cross over into the twenty-first century has not yet been created. Unless we adopt some new design techniques, we will be building time bombs into all of this new software, set to blow up at some time around the turn of the century. If everyone in the computer world begins to adopt these techniques over the next few years, there will be no need to worry about the conversion methods already discussed.

The first technique is so obvious that it hardly seems to be worth mentioning, but it is the key to solving the whole problem. We must begin to use four-digit years! The cost is two bytes per date, a small price to pay for extra storage when compared to the cost of converting software. Sometimes these two bytes can fit into wasted space at the end of a block of records, requiring no additional storage. Even when that does not work, it will take a half million dates to use up an additional megabyte of disc space, with a thousand dates fitting into the same space as a 25-line editor text file. Most importantly, the amount of effort required to begin coding in this manner is very small.

The second technique involves adopting formats that will represent dates unambiguously. This brings us back to one of the more subtle problems associated with date handling even today, which comes up occasionally at an international conference such as this one. Computers aside, those of us from the United States are used to writing our dates in the format Month/Day/Year. In many other parts of the world, a different order is used: Day/Month/Year. This can lead to confusion about the meaning of a numeric date on correspondence. Introduce computers and the many organizations that use them across international boundaries, and we are certain to have confusion when a date such as 03/02/86 appears on a report. Is this March 2 or February 3? There are 132 days each year when this problem occurs (12 months times the first 12 days of every month, less one day per month when both month and day are identical, such as 03/03). Each of these dates has two possible interpretations, but at least there is no confusion about the year since there is no month or day numbered 86. In fact, at no time since the invention of the computer has it been possible to confuse the current year with any month or day.

Now let's introduce the twenty-first century, which actually begins with 2001. This is about the only area where a two-digit representation for the year 2000 will not give us grief, since year "00" (2000) does not conflict with any month or day. But the year 2001 will cause a whole new round of confusion if it is displayed with only two digits. Sometimes computers use not only the two formats discussed above (Month/Day/Year and Day/Month/Year) but also Year/Month/Day, the order in which dates are often stored internally for sorting purposes. So how do we interpret 03/02/01? As March 2, 2001 (Month/Day/Year)? February 3, 2001 (Day/Month/Year)? Or February 1, 2003 (Year/Month/Day)? The number of ways to confuse dates increases significantly during the first 12 years of the new century, since the values "01" through "12" can now represent the month, day or year. There are 1716 dates where this confusion can occur during these twelve years (12 months times the first 12 days of each month times 12 years, less one day per year when all three are identical, such as 01/01/01). Each of these dates now has up to three possible interpretations.

This problem should be taken care of now by choosing unambiguous formats for displaying dates. First of all, the year should be displayed using all four digits, such as "2001". This can be phased in gradually between now and the turn of the century, since the two-digit year does not confuse us yet. Then we must find a way to distinguish the month from the day. One method would be to use ordinal numbers, such as 1st, 2nd, 3rd, to represent the day of month. These representations often differ across languages, so this does not solve the international problem. Since people rarely use them in correspondence, they might be awkward to computer users, and they could still possibly be interpreted as months. Another approach is to use alphabetic representations for the months, such as Jan, Feb, Mar. These are more familiar to people, and they cannot be confused with the year or the day. The problem of language differences still exists, but these abbreviations could be translated into the local language for displaying. A common set of routines, similar in concept to HP 3000 Native Language Support, could be implemented to take care of this localization.

The resulting dates have a format like "MAR 02, 2001" or "02/MAR/2001", which are both much clearer than 03/02/01. The important point is not what format we choose, but that every user understands exactly which date we are representing.

The third technique that will greatly aid our system design is the use of a productivity tool called the data dictionary. While dictionaries do not represent a new concept, many software applications are still being developed without them. A data dictionary maintains information about all our data elements, including which programs and data sets refer to each element. If we must expand a six-digit date by two digits to accommodate the year, the dictionary simplifies the conversion process by locating all the places where changes must be made. I predict that its value as a productivity tool will continue to be demonstrated by increasing popularity over the next few years.

Future Productivity Tools

The data processing world needs a whole new way to look at date-related information. Currently, dates are usually treated like any other piece of data containing six numeric characters, but in reality they have several unique properties:

1) Dates are rarely printed in the same format as they are stored. Slashes are often added, requiring program logic or an edit mask for every date field.

2) Dates must be stored in Year/Month/Day order for ease of use in sorting and comparisons. Since this is not how most users prefer to view their data, editing logic of some kind must be coded to rearrange the order.

3) There can be ambiguity about what a date field represents, depending on the order in which it is presented. As we have already seen, this ambiguity will be even greater in the next century when two-digit years such as "01" can be confused with the numbers representing months or days.

I propose that all future programmer productivity tools incorporate a special data type for dates. This data type will be handled within a data dictionary just like integer or character data is today. Every other tool that accesses dates, such as screen handlers, programming languages, code generators, and report writers, will be based on the data dictionary. Since all these tools will understand the meaning of the type called "date", they will all handle dates in exactly the same way.

The key here is that the internal storage technique can be completely isolated from the way users see dates. There will only be one way dates are stored internally, and every tool which accesses dates will understand that storage method. The technique must allow for the turn of the century and beyond, implying the ability to handle four-digit years, and for easy comparison between dates, implying an .ascending order. One example of such a technique would use 32 bits as follows:

High-order	16 bits:	Year	0 to 9999
Next	8 bits:	Month	1 to 12
Low-order	8 bits:	Day	1 to 31

Another example would start counting at some set date in history and add one for every day. We could also use eight ASCII bytes. While techniques like these are not new, the problem up until now has been that programmers needed to know the internal representation. This will no longer be true with our proposed productivity tools! No one accessing a date field will need to understand the internal storage method; the outer layer of tools will manage that for us. Some languages today automatically convert numeric data to a readable format for displaying; dates need to be treated the same way.

For this to work, all access to these date fields must be handled by the tools associated with the data dictionary. One objection to this technique has been that generic tools (such as QUERY/3000) do not understand the special meaning associated with these fields. In the future, generic tools must know what type of data they are accessing, beyond just character or integer. To accomplish this, all tools will need to use data dictionaries for their description of data elements.

The end result of providing these tools is that programmers will not need to worry about handling dates at all. Once a data element has been defined with a type of "date", the computer will take over with a set of common routines that understand both the internal storage technique and the default display formats for dates. The software developer or end user only needs to refer to the data element, and the computer will handle:

Sorting and comparisons
Editing (for valid dates)
Range checking (such as dates between March 1, 1986 and March 31, 1986)
Displaying in the proper format

This represents another step toward making computers friendlier to everyone who uses them.

Conclusion

Although we still have plenty of time to implement all of this by the year 2000, the years will go by faster than we realize. We must start thinking along these lines now. Those who develop productivity tools should begin to plan how their software will handle the arrival of the twenty-first century. And those who create software using these tools should begin to ask about plans for the future. It is not yet too late by any means, but as the British poet Andrew Marvell once said,

"At my back I always hear
Time's winged chariot hurrying near."

The time will creep up on us sooner than we think. So let's be ready, because on Monday morning, January 3, 2000, no one in data processing will be able to say that they were not given enough notice to solve these problems!

Biography

Kevin Cooper is currently a Software Engineer (SE) with Hewlett-Packard Company, Neely Sales Region, supporting HP3000 installations in the San Francisco Bay Area. Prior to this he worked for eight years as a programmer/analyst and project leader in Hewlett-Packard's internal Information Systems organization. He holds a bachelor's degree in Computer Science from the University of California, Berkeley.