Year 2000—Making Good Time

by George James and Jon Diamond

Where are we now?

Approximately halfway through the year 2000 crisis, the majority of organizations have by now taken stock of the situation and have plans in motion to address their own problems. The picture is not, however, totally uniform. Large organizations are further forward than smaller organizations - they generally have more to do and consequently have needed to start earlier. Some countries have achieved greater awareness than others - the US, UK and Australia have significant government programmes that have informed and stimulated action, France and Germany have given greater priority to preparations for European Monetary Union and the Far East has been pre-occupied by local economic troubles.

Much therefore is being done right now, but there is still much to be done. Many organizations have yet to address their year 2000 problems. While they are at a disadvantage by having less time they will at least be able to benefit from the experiences gained by others who have already paved the way.

This paper imparts some of the lessons that have been learned from running year 2000 remediation projects and in particular looks at some of the specific problems and issues to be found in the assessment and remediation of M applications.

What have we learned about year 2000 remediation projects?

In the past two years we have assessed and corrected over two million lines of M code across a range of different applications of different vintages and coding styles. In the course of these projects we have learned a great deal about the various technical and project management issues involved in year 2000 conversions. Some of these are summarized below:

- Watch for software falling through the cracks

The need to create an inventory of all systems and applications within an enterprise is well established. In large organizations that have many systems it is very easy for small but important pieces of software to fall through the cracks. Operating system command files (for system start-up, shutdown, backups, etc) and end-user-developed spreadsheets and PC applications are typical of this type of oversight.

- Do not believe what the experts say

In the planning stages of a year 2000 project we always ask the customer what their Event Horizon is. This is the date on which the system will fail if no action is taken. It is important to plan the project so that it will be complete before the event horizon.

We have discovered that the customer is often not the best judge of what the Event Horizon actually is. When asked they will often give a confident statement like 'Budgets are drawn up in October for the following calendar year and so our event horizon is 1 October 1998'. The implication is that they know about all date processing within their system and it does not deal with any dates more than 15 months into the future.

Even if the customer has intimate knowledge of their application, this statement should be interpreted to mean that the event horizon is 1 October 1998 at the latest, but could be earlier. We have learned to perform an exhaustive analysis of the database in order to identify possible Event Horizons that are otherwise unknown.

- Do not believe what the experts say (again)

'There is no need to worry about that application, it was only developed last year and so it is compliant'. Unless an application has been assessed and tested, then any such claim is groundless.

I have lost count of the number of times I have seen the following statement made in the comp.lang.mumps newsgroup: 'My application was developed using FileMan. FileMan is year 2000 compliant. Therefore my application is compliant. When the US Department of Veterans Affairs performed an assessment of their FileMan based applications they found that across 21,000 routines about 8% required some change. Clearly this syllogism is not true.

• Do not ignore the User Interface

It is very easy to make the assumption that the user interface must be modified to force the user to enter dates using a four-digit year and for all output to be displayed using four digits. The reason that dates are entered in two-digit year form is to reduce keystrokes. This is generally a good thing. (Internal date representation using two digits is generally a bad thing). Likewise the display of dates using a two-digit year is primarily to maximize the use of screen and report real estate. It is only where the date has a large range (e.g., date of birth) that four digits are actually necessary for data entry.

So it is important to allow the user to enter dates using two digits. This implies that a windowing algorithm must be applied to the date as entered to expand it to an unambiguous form.

• Inspecting two million lines of code

A year 2000 problem could have been coded anywhere in your application. To be sure, you have to inspect every line of your application.

For a badly written application with intensive date processing we have found that we need to change, on average, as much as one in fifty lines of code.

For a good application the problem is more akin to finding a needle in a haystack. We have inspected applications that have a rate of as little as one problem per one thousand lines. The challenge here is how to find these problems in a cost-effective manner.

The scale of the problem demands a systematic and automated approach. Our RE/2000 code-scanning tool is ideal for this purpose. It searches code and highlights any syntax that is indicative of date processing using a color-coded scale of severity. It can also be tuned to cater for application-specific coding styles and idiosyncrasies. Our programmers using this tool can achieve a productivity, consistency and thoroughness of problem identification that is many times higher than is possible without automation.

• Code conversion

Surprisingly we have found very few cases where automating the actual conversion of the code gives much benefit. However, some applications have commonly repeating fragments of code that can be converted en-masse. The major benefit of such en-masse conversion is simply the removal of typographical errors from the code editing process. This type of conversion can mostly be performed with simple routine search and replace tools. More complex transformations can be achieved using a syntax sensitive tool such as RE/parser if necessary.

• Checking all those changes

Unit testing and system testing are essential parts of the quality control process. However, because of the relatively sparse nature of the changes that are made and the fact that unit testing and system testing have a very functional basis, the defect yield from such testing is very low. In other words, a high number of testing hours are required to find a small number of bugs.

In many of the applications that we have converted we have found that the best yield is achieved from desk checking of converted code. Use of a highly experienced programmer or consultant to check marked up before-and-after listings can identify ten times more defects per man hour than the same effort invested in functional- based testing.

• System testing

It is generally agreed that a converted application needs to be tested both with contemporary dates (to ensure that the application continues to work correctly today) and with a range of future, post-year-2000 dates.

Eight Lessons from the Front Line

by Dan Looper

The approaching millennium date change must be the most frustrating, annoying, frightening and discussed milestone that the Information Technology user has ever experienced. What about the lessons that we are learning? Will they be forgotten beyond this date change?

Lesson one: Sizing your challenge is difficult. There are no magic formulas for sizing the year 2000 problem. Instead, it is an iterative process of planning, executing, learning and planning again. Many companies need a partner to accomplish the task. Select a partner, assign tasks, not blame, and get started.

Lesson two: Avoid trial and error. Much time can be wasted by changing approaches in mid-stream. Assess the methodologies and decide upon an approach that is to be taken. Establish it as the only legitimate one for the enterprise, and cease trying the "silver bullet."

Lesson three: Accept cost realism. Whether the cost to repair/replace our applications is $1.00 or $10.00 per line of code should not be the primary concern. One dollar of correction today could save us $4.00-5.00 between now and the Year 2000. Recognize and acknowledge the true costs as that of lost revenue.

Lesson four: We don't have all of the answers. Many large companies have taken critical systems and subjected them to a rollover to January 1, 2000. The results have been severe. We must learn to say, "I don't know all of the answers," and let this become a driving factor in building and implementing technology as a support tool for our businesses both now and beyond the Year 2000.

(continued on next page)

Lesson five: Correct limited focus. We tend to become so tied up with looking at applications that we forget such things as infrastructure, system interfaces, partners and suppliers, and contingency planning. Consider the entire issue at hand rather than focusing on just the applications. Applications alone will not allow businesses to continue to function.

Lesson six: Tools don't fix the problem, people do. There are many good tools on the market. There are also many "Master Mechanics" both within the companies experiencing the problems as well as through partners; use the mechanic.

Lesson seven: Testing as usual. Testing must begin as a risk abatement process when we are inventorying our technology. A test readiness review early in the process should be used to determine whether we can realistically hope to test both function and date handling for Year 2000.

Lesson eight: Honesty is the best policy. We know what must be done to avoid catastrophe, yet we are still telling ourselves the ultimate lie; "This problem just cannot be this bad." Eight years of "doom and gloom" have had little impact. Why? We have great difficulty being honest with ourselves. There is a belief that we can short cut the process of discovery and repair. Be honest, admit that there is a problem, and tackle it head-on.

With less than 600 days remaining to correct the problem, we should be planning for our New Year's Eve 1999 celebration. Be diligent! Efficiency and saving unnecessary spending must be the goal to which we all subscribe between now and December 31, 1999.

Dan Looper is Year 2000 National Account Manager for Litton/PRC. He can be reached via email at: Looper_Dan@prc.com

Contemporary date testing can be performed by using a copy of a production database and comparing the results directly with those obtained before conversion. This is a simple parallel run scenario.

Future date testing is more difficult. In many cases small amounts of test data can be created for future dates and the results compared against predicted results. However, this is often impractical where the application is complex and needs to be tested against a large amount of varying test data. It is also impractical when the application is driven by chronological data in the database (for example, outpatient appointment scheduling).

We have achieved large scale testing for future dates by automatically 'aging' the data from a copy of the contemporary date test database. With two databases that are identical except for date fields it is possible to perform a kind of parallel run between a contemporary database and a future database.

Depending on whether the processing cycles and reporting in the application are daily, weekly, monthly or annual it may be necessary to 'age' the database in increments of either 4 or 28 years (Why 4? To ensure that transactions processed on February 29, 1996 becomes February 29, 2000 and not February 29, 1999 or February 29, 2001 neither of which are valid dates. Why 28? Well, it turns out that this is the number required to ensure that all dates are on the same day of the week, with the same leap year structure as the original date).

If you know the location and format of all the date data in your system then you can do this mechanically, and fairly easily. However it is not a fast process and does require considerable hardware resources.

• How to make a time machine

The amount of hardware resources required to perform comprehensive testing of a large application with a large database can be quite a surprise. While the cost of hardware means that this really should be the least of your problems, the cost of additional software licenses and the logistics of hardware procurement can be formidable obstacles.

You should consider the need for multiple copies of your application and database. You should also consider that as time runs out you might have to run multiple system tests (possibly with different system dates) concurrently. While the easiest way to achieve this is to use multiple machines, this is not always feasible.

It is worth knowing that all InterSystems and Micronetics M implementations support the ability to artificially alter the M system date (i.e. $H) without altering the underlying operating system's date. For example, on a VMS machine you could configure two DSM environments, one with a contemporary database and $H value and the other with a future database and a $H value. Both these environments can be run simultaneously, enabling direct comparisons of the application in both the 20th and 21st centuries.

• How can you be sure you tested your application?

In one of our first year 2000 conversion projects we converted all of the code and then handed it over to the customer for acceptance testing. The users had been carefully preparing test plans and immediately set out to test the code we had delivered. After two weeks they reported to their management that they had completed their tests and were happy to accept the software.

How could their management be sure that the users had performed sufficient testing? Well, we were able to tell them! When we delivered the software we had instrumented it with a test coverage monitor (a feature of the RE/2000 tool). Their management was able to examine the test coverage statistics for the application and assure themselves that the acceptance testing had been thorough and comprehensive.

The sparse nature of changes that span the whole application makes it very important to be able to measure in some quantifiable way the extensiveness of any testing that is performed. A test coverage monitor is a simple and effective way of achieving this.

What have we learned about the M problem set?

In some ways M systems have fewer problems than other systems since there is no standard format for dates which is supported by date-processing primitives within the language as there is in COBOL. The format of $HOROLOG has, on the other hand, influenced many developers to store dates based on the $H range within the database and do processing/calculations on this format.

In general this is good. However, even when the $H format is being used there are many times when calculations need to take into account months or years, which imply calculations on the $H values to produce, for example, the number of months between two dates. More commonly, end-users are somewhat uncomfortable with the $H format (!) and demand that dates are presented to them and input by them in formats that they understand. These conversions are also not supported directly by the M language.

This means that any problems which are encountered are mainly homegrown by the specific developers of the system. However, in more recent times the vendors have been helping users by providing date conversion utilities which make the conversion process run faster, handle multiple (international) date formats, etc. These are primarily the $ZDATE and $ZCALL(%CDATASC,) function set.

Unfortunately, like most software, these have developed over time and the specification of them has changed subtly, but without necessarily the users of these functions being aware of the changes. For example, at one time some of these functions did not handle 21st century dates at all, then they produced results with two-digit years for 21st century dates, then results with four-digit years for 21st century dates (but only two-digit years for 20th century dates).

This would not necessarily matter too much if the only use for these functions were the display of dates on a screen. (Users are very flexible and can determine what these subtle changes mean, and the impact on the display layout).

However, these functions provided a general capability and so were used in that light. Frequently the users of these functions made assumptions about the format of the results and have used the results for further processing. For example, this happens if the year part of the output from $ZD has been used in a fixed format interface file (two-digits only). This will now fail for dates in the 21st century.

Ad-hoc Date Errors

Programmers can be very adept at overcoming problems. Unfortunately they do not always realize the implications or test what they have done thoroughly enough. Some real-life examples of coding problems that have been encountered in practice are:

• The calculation of number of days in a year:
```
I $E(YYYY,3,4)=”00” Q 365
```
This fails in the year 2000.

• Adding 1900 to a two digit year:
```
S FULLDATE=1900+$E(VERDAT,1,2)
```

• Embedded assumptions that the dates can only be in the 20th century
```
W 19,$P(DATE,” “,3)
S YY=”19”_$E(X,1,2)
S DATE=”05APR19”_YEAR
```
This last one gets the last day in the U.K. tax year!

• Calculation with two-digit years:
```
I $P(X,”/”)+1’=$P(X,”/”,2) ...
```
and
```
I CY>(X-2),CY<(X+2) ...
```
The user had been prompted to enter a year range into the variable X as YY/YY. This works happily for 97/98, but not for 99/00.

• Just bad validation code:
```
I X’?1”19”2N,$E(X,3,4)> ...
```
This will fail in the year 2000 since 00 is not greater than 99 (which could be the right-hand-side of the comparison).

Other examples of ad-hoc errors which have been observed in code:

• Incorrect validation of 2000 as a leap year.

• The day number of the last day of a leap year being returned as 365, rather than 366. This is a non-obvious but very common logic error in date algorithms. The last day of a leap year should be an explicit test case whenever testing date algorithms.

• Dates beyond a certain date being invalid, e.g., 31st December 2049, but not consistently so throughout a system.

• Inconsistent windowing. The following examples were found in different places within a single application:

```
S DATE=$S(X<90:20,1:19)
and
S CC=19 S:YY<50 CC=20
```

External Interfaces

Whilst an individual system can be compliant it is unlikely that, in most organizations, a system will be in isolation. Any compliant system is therefore likely to need to pass date information to and from other applications. Since most of these will be non-M ones, then dates are almost certainly going to need to be passed in non-$H formats, typically with a two-digit or four-digit year.

In most organizations it is unlikely that all applications which are made compliant will be implemented at the same time, as a big bang. So each application or a small number will need to be upgraded at the same time (a package). The interfaces between each of these applications within this package can be changed to make the date explicit, typically using a four-digit year. However, interfaces to systems outside the package are problematic. It may be possible to change the other end of the interface at the same time, but as the number of interfaces increases this becomes a more and more theoretical approach.

Most interfaces will therefore have to remain the same, probably using a two-digit year. It is notable that international EDI standards have only recently allowed four-digit years in messages, and therefore all EDI-based systems are likely to continue using two-digit years for a long time, if not forever. Conversely, it is notable that the US government has mandated that all interfaces between government departments (but not within departments) must use a full four-digit year.

Use of a two-digit year can work well provided that the window used is the same on both sides of the interface. However, you may not have access to or control over the window algorithm in all your applications, such as third party packages. (Even Excel, Access and other Microsoft products have their own pivot dates that have changed between versions!)

Therefore you should exert considerable caution in this area, documenting precisely what is being assumed, and perhaps use a parameterized window which can be changed interface by interface, rather than using a standard one for the complete application. The U.K. retail industry has decided to use a date window of 1950-2049, which should make things simpler for them, but even this diktat won't necessarily mean that every package/application can be changed to conform to this.

Note: One very frequent use of dates in an interface is embedding the date in a file name, such as in YYMMDD format. Unfortunately, due to the DOS filename restrictions of 8+3 characters, it was often impossible to create/use files with four-digit years. Thus these files could have potential problems, either of collation within a directory, or incorrectly derived year value, etc.

It is thus likely that many problems will not occur within an application, but at the interfaces between them.

Database problems

• Date Storage

Dates stored using the $H format/range are compliant, but often many systems will use non-compliant formats for a number of reasons.

Some of these reasons include:

• Historical

The system may be so old that none of the designers thought about the 21st century, since the lifetime of the system was probably 5 years, 10 years maximum. (There are numerous systems still running which were originally built in the 1970s or early 1980s).

• Performance

Older systems especially had to be more concerned about CPU power and the processing overhead of date conversion was very high if significant amounts of input/output were performed.

• Convenience

For example, the date is only output and not used for further processing. (This is often an assumption that is overtaken by subsequent events, however!)

• Because the $H format is inappropriate

For example, a date which only represents a year, month, accounting period, etc.

• Support

It is much easier for a support programmer to interpret a YYMMDD date in a database (e.g. 980420) than a $H date (e.g. 57452).

• Non-$H storage formats

Some possible storage formats for complete dates used in practice are DDMMYY (international usage), MMD-

DYY (US and Canada), DDMMMYY (with a three character month name dependent on system language) and YYMMDD. When data is stored in nodes using these formats the system may be perfectly acceptable, provided a windowing algorithm is used within the application for appropriate interpretation of what the year means.

For example, dates stored as 010160, 010110 and 010125 using a window of 1920-2019 would be interpreted as 1st Jan 1960, 1st Jan 2010 and 1st Jan 1925 respectively.

However, if the window is changed at some stage in the future, because the system needs to be able to process dates further in the future, then these formats may cause problems if old data is still held within the database. Again using the first example above, moving the date window to 1950-2049 would cause 010125 to be interpreted as 2025 and not the original interpretation.

In many cases partial dates are also stored in a database, and these can be particularly difficult to spot without an intimate knowledge of the database structure. Examples of these are frequent in Accounting systems, such as YY as accounting year or YYMM as accounting period.

Windowing can also resolve many of the usages of these types of dates. However, the problem with many of these formats becomes much more severe when they are used in subscripts, typically with a leading two-digit year. This occurs for three reasons:

• Some software producing, for example, YYMMDD dates may only produce either a one or two digit year for 2000-2009 (i.e., 50101 or 050101 for 1st Jan 2005) depending on the way the subscript value is created.

• Calculations on two digit years may not wrap around the end of the century (i.e. subtracting 1 from 00 should produce 99; adding 1 to 99 should produce 00).

• Dates in the 21st century will no longer collate after dates in the 20th century and most systems will have some functions which `$ORDER` on dates. (Actually, because strings collate after numbers in M globals the years 00 through 09 will collate correctly giving rise to a possible 2010 problem if not picked up now).

Existing Date Problems

Another problem that may be hidden within a database is the usage of dates by users, or systems, to have special meanings. For example, many systems will have dates of 31st December 1999 in an end-date field to mean effectively that there is no end. Some systems using DDMMYY or YYMMDD formats have also used 999999 as a special value.

Some actual uses have been Value Added Tax rates and Branch closing dates using 31st December 1999 as an end-date. The VAT date was entered by users in order to overcome the problem of not being able to specify 21st century dates and not reported to the systems developers as a problem since the users had a work-around. Unfortunately the work-around only had a limited lifetime! The Branch date was a similar problem, but this time it was partially created by the system developers since they embedded this date in the user-enterable format (31DEC99) within the application.

In the case of this latter system, the standard is to use two digit years everywhere and use a windowing algorithm on input/output. Unfortunately, due to past programming and user problems, dates which pre-date the low end of this window are stored in the database (sometimes in the OK $H format). It is therefore not clear what impact these dates will have on the system, even when it is Year 2000-compliant. So, these will need to be located and eliminated as soon as possible.

Conclusion - Guilty until Proven Innocent

For M applications some of the issues which concern other development environments are not applicable. Nevertheless, we expect that most, if not all M applications will have problems which need to be rectified. Some applications which have embedded dates in their database structures in non-$H formats could have a significant amount of remediation work.

Our experience shows that, even for a well written application, there are a number of subtle problems which can be quite difficult to locate, or test for. Given these issues, visual inspection, together with an independent verification and the assistance of appropriate tools has proved to be more beneficial than other techniques in locating and fixing bugs.

Testing is obviously important and requires a number of new ideas to be introduced—rolling forward of historic data and system dates, changing of date windows etc. Implementing these ideas is potentially error-prone, so automating (and testing) them is also vital.

It is often quoted that "if it uses electricity, then it is guilty until proven innocent." Applications written in M may be less guilty than others but the principle still applies. Even if it's written in M you still have to prove it's innocence. **M**

Jon Diamond is an independent consultant. He can be contacted at jdiamond@btinternet.com.

George James is Managing Director of George James Software. He can be contacted at georgej@georgejames.com.