



YEAR 2000 White Paper

The Information Systems community is racing toward a failure of its standard date format. Today, many applications at the core of corporate data use a 2-digit year format that will not meet the requirements of the YEAR 2000 (Y2K). The widespread use of 2-digit date fields has created an enormous challenge. Virtually all computer-based applications depend on some date processing. The potential impact of the approaching century changeover could be devastating.

Applications that only deal with 2-digit years will not be able to distinguish years before and after the century changeover. Simple math: Does 00 come before or after 99? This misinterpretation will cause a variety of calculation, reporting, and decision errors. The YEAR 2000 threatens the integrity and performance of enterprises dependent on legacy systems all over the world.

A common misperception is that the YEAR 2000 problem arises at midnight, December 31, 1999. Some applications have already begun to experience malfunctions related to the YEAR 2000. In some cases software programs cease to function, while in other cases software programs produce erroneous data and results. The decision making process is vulnerable if data is not correct. It doesn't take much to imagine the potential damage: licenses and permits not issued; payroll checks not cut; inaccurate personnel, medical and academic records; errors in banking and finance; accounts not paid or received; inventory levels not maintained; and weapon systems not functioning properly.



Published by:
Thoroughbred Software International, Inc.
19 Schoolhouse Road
PO Box 6712
Somerset, New Jersey 08875-6712

Copyright © 1999 by Thoroughbred Software International, Inc.

All rights reserved. No part of the contents of this document may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Document Number: WP002

The Thoroughbred logo, Swash logo, and *Solution-IV* Accounting logo, THOROUGHbred, IDOL, OPEN WORKSHOP, AND VIP VISUAL IMAGE PRESENTATION are registered trademarks of Thoroughbred Software International, Inc.

Thoroughbred *Basic*, *Thoroughbred Environment*, *OPENworkshop*, *IDOL-IV*, *Dictionary-IV*, *Script-IV*, *Report-IV*, *Query-IV*, *Source-IV*, TS Network DataServer, TS ODBC DataServer, TS ORACLE DataServer, VIP (*Visual Image Presentation*), VIP4, GWW (*Gateway for Windows™*), *GWWImage*, TS ChartServer, TS ReportServer, TbredComm, WorkStation Manager, and *Solution-IV* are trademarks of Thoroughbred Software International, Inc.

Thoroughbred Software International, Incorporated licenses the U/SQL Client-Server product from Transoft Limited. Thoroughbred Software International, Incorporated incurs sole responsibility for support and maintenance of the Thoroughbred U/SQL Client-Server product. Transoft, AIM, OEO, UBB, U/BL, U/FOS, U/Gi, and U/SQL are trademarks of Transoft Limited.

MS-DOS, Xenix, Windows, Windows for Workgroups, Microsoft Windows 95, and Windows NT are trademarks of Microsoft Corp. IBM, IBM PC, OS/2, PS/2, and PC-DOS are trademarks of International Business Machines Corp. DEC, OPEN VMS, and ULTRIX are trademarks of Digital Equipment Corp. UNIX is a trademark licensed exclusively through X/Open Company, LTD. Novell is a registered trademark of Novell, Inc. Oracle is a registered trademark of Oracle Systems Corporation. InstallShield is a registered trademark of Stirling Technologies, Inc.

Other names, products and services mentioned are the trademarks or registered trademarks of their respective vendors or organizations.

BACKGROUND

It took some time, but the industry has finally gotten past – *that date century bug*. Unfortunately, the fear of change, procrastination, and the reluctance to address the problem all exist. This is one problem where the deadline cannot be extended. The problem is very simple. In an attempt to save precious space and perhaps reduce the number of keystrokes during data entry, most applications allocate 2-digits to designate the year. These systems never expected to be in service at the turn of the century. The 2-digit notation system uses 2-digits to represent the month 01 through 12, 2-digits to represent the day 01 through 31, and 2-digits to represent the year 00 through 99. If this sounds all too familiar, you are not alone. Thousands of Thoroughbred-based applications in production at the turn of the century will have been developed before the industry became YEAR 2000 concerned.

The computer industry created a faulty standard. The standard in question is how we enter and store dates using software applications, usually MMDDYY or DDMMYY or some variation where only 2-digits are allocated for the year. Dates with 2-digit years can be found in millions of data files within millions of applications.

As long as the data is not ambiguous and dates are interpreted to be in the same century, applications will continue to function properly. That is until the year 99. Date manipulation, arithmetic, compares, and sorting are relatively simple functions using these dates. But what's the life expectancy of an application that can only handle dates from one century?

There are a number of alternatives when considering updating an application to handle YEAR 2000. But they all boil down to modify or replace. Many legacy applications have survived the 70's, 80's, and now the 90's. There should be no surprise when software developers attempt the YEAR 2000 hurdle using 3GL technology and once again preserve their past investments.

The YEAR 2000 should not be considered a '*bug*.' Developers fix bugs and move on. The YEAR 2000 requires a philosophical change to the way people abbreviate dates. The standard of specifying a year using 2 digits has been an accepted practice for a long time. For the computer industry, it has been a lifetime. It is time for a new standard. The good thing about standards, there are so many to choose from. There are numerous standards, both public and proprietary, for providing YEAR 2000 readiness.

THOROUGHbred SOLUTIONS

Thoroughbred's development environments *OPENworkshop*[™] and *IDOL-IV*[™] provide automated solutions to the YEAR 2000. Both environments are based on *Dictionary-IV*[™], which provides a number of date choices that support beyond on the YEAR 2000. Thoroughbred *Basic*[™] developers can also interface with *Dictionary-IV* taking advantage of the date choices. All Thoroughbred development environments (including 3GL) have access to SQL dates; one of a number of the industry accepted YEAR 2000 solutions.

The SQL date provides a standard method for storage, compares, sorting, validation, and arithmetic. The SQL date is a number representing a count of days from some starting point – typically January 1, 1. The count must be accurate accounting for leap years. For example the year 1900 is not a leap year but the year 2000 is a leap year and includes February 29. How many digits are we talking about? Some simple math shows 365 days x 2000 years = 730000. Dates between January 14, 274 and November 25, 2738 are represented by 6 digits. This fact is instrumental when considering the conversion process of a non-Y2K ready database.

Identify the Development Environment

Applications developed by the Thoroughbred VAR community comes in many different flavors. Many are a collection of Thoroughbred *Basic* programs with some form of structure but not conscious of *Dictionary-IV*[™]. Some Thoroughbred *Basic* applications have incorporated integration with *Dictionary-IV*, while others have simply exposed selected data file layouts. Some Thoroughbred developers are using a hybrid of Thoroughbred *Basic* and *Script-IV*[™] code, while still others are completely into *OPENworkshop* and/or *IDOL-IV*. Each application developer must make an honest determination of where their development environment lies:

- Thoroughbred *Basic* only
- Thoroughbred *Basic* with some *Dictionary-IV*
- *OPENworkshop* or *IDOL-IV*

Dictionary-IV update

Dictionary-IV version 8.4.1 supports a new date type as well as extensions to existing date types. The 8.4.1 date type updates are key tools for providing Y2K support and readiness. Date types 2 and 7 have been enhanced to support years greater than 99. The new date (type 8) stores the Thoroughbred *Basic* SQL date as a number in your data files. This storage method removes a number of problems caused by the binary storage found with date type 5.

Date type 2 (stored as YYMMDD) continues the counting pattern using alpha characters. A0 follows 99 and B0 follows A9. This date type requires a PRM BASE-YEAR. The BASE-YEAR= parameter is used to specify the base year for 00. Typically, this value is either 1900 (default) or 1800. The following table shows the storage of the year for both 1800 and 1900 base.

Year	Base=1900 Stored as	Base=1800 Stored as
1900	00	A0
1990	90	J0
1998	98	J8
2000	A0	K0
2010	B0	L0

Date type 7 stores the year as CHR(YY+1). This allows the storage of years up to 99. The storage formula has been changed

```
from:   CHR(YY+1)+CHR(MM+1)+CHR(DD+1)
to:     CHR(MOD(YYYY,1900)+1)+CHR(MM+1)+CHR(DD+1)
```

The new date type 7 formula supports both 2 and 4-character years. But as a compressed date, it will eventually generate a binary value that may be misinterpreted. Using this technique, the year 2037 is stored as a field separator (\$8A\$).

Date types 2 and 7, while providing a quick fix, still rely on data compression and/or logical interpretation of data. Thoroughbred *Basic* has long supported a date type suitable for YEAR 2000 development. The SQL date is an industry accepted Y2K solution providing automatic functions for sorting, compares, arithmetic and storage. The Thoroughbred *Basic* SQL date is now available to *Dictionary-IV* developers in a new date type. The new date type stores the Thoroughbred *Basic* SQL date as a decimal number. The format storage length depends on whether you require time.

The input and display for the updated and new date types are determined by Global variables. The input and display can be a combination of 6 and 8-character formats. If an application's dates are not ambiguous in regard to century, displaying 6 may be sufficient. The Global variables also provide for specification of a *Date Method*. The *Date Method* provides the opportunity to modify the users input prior to date validation. The most common use of the *Date Method* is to determine and add a Century value to a 6-character date. For example; the user enters 082859 and the input mask is set to MMDDYYYY. Normally, a date validation routine would fail this input. The mask is 8 characters but the user only entered 6. With *Date Methods*, your application can now supply the missing information prior to validation by adding the proper century value. The *Date Method* has access to all kinds of information to assist in determining the Century including the dates data-name, format and link names. Thoroughbred provides a sample *Date Method* that replicates the Thoroughbred *Basic* sliding 50-year rule to determine the century. The method can be maintained as either a Thoroughbred *Basic* program or an *OPENworkshop* script method. This method is provided to help assist developers in providing a solid Y2K solution.

OBJECTIVE

Your game plan needs to begin with a simple objective: *Implement a common date storage that supports Century*. This provides the foundation for program logic (compares, validation, arithmetic), reporting (display formats), and user interface (input and display formats) to support 4-character years. Your solution must:

- store and display dates in a way that is not ambiguous to the determination of century
- correctly accept, recognize, calculate, compare, sort, and process dates they may span multiple centuries
- not abnormally terminate or produce erroneous results as a result of dates that span centuries
- support display and end-user interfaces requiring 4-digit year data
- calculate and process leap year information (i.e. 2000 is a leap year but 1900 is not)

What do you fix – data or program code? The reality is that both will be effected but data should be the primary concern when dealing with the YEAR 2000 problem. You can think of the program code as a vehicle for manipulating data. Without the data, there would be no need for the code. If the data is correct and the code is bad, it's patchable. If the data is not correct, it really doesn't matter what the code is doing.

A date can be defined in many different ways. The date definition includes the storage format, the format users input, display format for inquires and reports, the format used for I/O, and the format used by program logic for validation, compares, and arithmetic. Understanding each of these formats is not always a simple task. The dilemma becomes even more complex when the date field is built into the logic of legacy programs.

Again, your objective is to implement a date storage that supports century. All program code changes, display/input formats, and file I/O will be based on your new date storage. So, what are your choices for date storage?

- *physically store 8 characters YYYYMMDD*
If you are starting from scratch, this storage method may be appropriate. It does not simplify date validation and arithmetic. Both will still require coding. The biggest obstacles are your legacy data files. You are now storing 8 characters rather than 6. Record sizes will need to be expanded. Key fields that include dates will need expansion as well as any alternate index or sort definition that includes dates.

- *logically interpret using 6-character format*

This solution buys some time. One method is to apply a sliding 50-year rule. This doesn't effect the storage. It does require code modifications to compare, sorting, and arithmetic. The biggest downside is that the logical interpretation of data may be different from one day to the next. As the 50-year rule slides, so does the value of data affected. A second method employed by some developers is a notation that represents dates after 99. For example, A0 through A9 would represent 100 through 109 and B0 would be 110. Assuming 99 represents 1999 then A0 would represent 2000 (see the table on page 2). Though this solution solves the 6-character storage dilemma as well as sorts and compares, it introduces another level of application coding accounting for inputs and date validation.
- *data compression*

Implementing a date storage method that requires 6 spaces does simplify the Y2K conversion process. But, is data compression the answer? Are we still trying to save bytes of storage? This method requires 'compress' and 'expand' algorithms. Chances are the compressed data may include binary values that get misinterpreted by Thoroughbred *Basic* and/or your application. Values can be misinterpreted as field separators (\$8a\$), quotes (\$22\$), and nulls (\$00\$) to name a few. Most, if not all, Thoroughbred *Basic* (3GL) legacy applications were developed using IOLISTs. A binary storage is not a viable solution in an IOLIST environment.
- *SQL day-counter*

This storage method provides the ability to store 6 characters into an existing data record where 6 characters are currently being used. Therefore, there are no expansions required for record sizes or key sizes. The storage uses decimal numbers without any data compression. The SQL number is also an industry standard with common functions that accomplishes validation and 6- or 8-character date to SQL number conversions.

SOFTWARE INVENTORY

How do you start to solve the YEAR 2000 problem? Before diving into a project as substantial as YEAR 2000, you will need to inventory your software. Without a good inventory, the YEAR 2000 problem will take on the characteristics of an onion. As you peel away layers of problems, you find more layers of problems underneath. And the more layers you peel the more you feel like crying. The goal with software inventory is to quantify the number of date fields that make up your application. From there, you can start to examine the number of programs and data files affected as well as other application objects like screens, reports and tables.

Someplace there exists an application dictionary with file layouts. With legacy software this may be hard-coded within programs, documented off-line, or a file containing simple file layouts. Depending on the age of your dictionary source, there may be a question of accuracy. If your file layouts are not in a dictionary format, now may be a good time to consider *Dictionary-IV*. The correctness of your data and data definitions are essential for a successful YEAR 2000 migration. By defining your definitions in a centralized repository, you ensure the integrity of data throughout. In addition, the dictionary approach makes life easier by reducing the burden and cost associated with future software maintenance.

With the file layouts collected, you should be able to identify which data elements (or data names) are dates. Most, if not all, dictionaries include some data definitions for dates. This can be in the form of a date indicator or date type. With this information you should now have a better handle on quantifying the number of dates that makeup your application. How well do you really know your application? Are your dates all stored in the same format (i.e. YYMMDD)? Are all your dates displayed and entered in the same format? Many dates will be used in multiple files by multiple programs. Again, *Dictionary-IV* can be used as a valuable asset in providing Global Cross-referencing. Performing this exercise using the BAS A/R Accounting Module, we were able to determine the following dates needed to be addressed.

YEAR 2000 White Paper

Data Element	File(s)
AR_ADJ_DATA	Adjustment Journal
CUS_INV_DATE	Invoice Journal, Open A/R
CUS_LPYMT_DATE	Delinquent Age Trial Balance
CUS_PYMT_DATE	Open A/R, Cash Receipt Journal
INVOICE_DATE	Sales Commission
JOURNAL_DATE	Non-A/R Cash Receipt
LARGEST_BAL_DATE	Customer Master
LAST_PAYMENT_DATE	Customer Master
SVCG_TRAN_DATE	Statement Service Charge

Once quantified, the next step is to determine where and how each date is used. Solving the 2-digit date problem is not technically difficult. Most tasks can be managed with careful planning. Before you start any program coding, you will need to be able to fill out the following form.

Data element	Storage format	Display format	File(s)	Where used/type
1.				
2.				
3.				

THOROUGHbred BASIC PROGRAM WITH SQL DATES

Many legacy applications deal with dates via substrings. If a date is in MMDDYY format, then X\$(1,2) contains the month, X\$(3,2) contains the day, and X\$(5,2) contains the year. All date manipulation, input/display and validation would be based on this format. Implementing SQL dates, not only solves the Y2K, but also simplifies the above process significantly. Before you start coding using SQL dates, there is some Thoroughbred *Basic* syntax that needs reviewing.

CDN is a Thoroughbred *Basic* system variable. It holds the current date value in an SQL numeric format. The granularity of CDN is controlled by precision. With precision set to 0, CDN returns today's SQL numeric value while a higher precision may include time (hours, minutes, and seconds).

DTN and NTD are Thoroughbred *Basic* functions used to convert dates to SQL numbers (DTN) and SQL numbers to dates (NTD).

```
PRECISION 0 ;
LET A = CDN ;
PRINT NTD(A, "MM/DD/YYYY")
```

Now suppose you want to validate a date input and convert to an SQL number.

```
INPUT "Enter date (MMDDYYYY): ", A$ ;
LET A = DTN(A$, "MMDDYYYY", ERR=BAD_DATE)
```

The DTN function solves date validation and conversion to an SQL number. But that's not all. DTN can also be used to generate values for century when dealing with 6-character date inputs. This is accomplished by using a sliding 50-year rule. This allows users to continue to enter dates using 6 characters. The sliding 50-year rule uses the current year as a midpoint for a century (100) worth of dates. All 2-digit years are within a range of 50 years in either direction. For example, in 1997 the range of years is 47 through 46 or 1947 through 2046. In 1998 the range changes to 48 through 47 or 1948 through 2047 and so on. Try the following in Thoroughbred *Basic*. Does it print 1945 or 2045?

```
PRINT DTN("123145", "MM/DD/YYYY")
```

So far, our examples have only used MMDDYY and MMDDYYYY for date masks. There are quite a few SQL date masks that will inevitably simplify existing Thoroughbred *Basic* code. In the above example, our concern was which 4-character year would print. In that case, the date mask should have been "YYYY" and not "MM/DD/YYYY."

Does your legacy application prevent due dates on Sunday? The Thoroughbred *Basic* code without SQL dates could be substantial. How about date arithmetic? To add 30 days to a date the Thoroughbred *Basic* code must take into consideration crossing over a month, year, leap years, and now centuries. Again, SQL dates simplifies the process. For example, assume an invoice date was entered by a user in MMDDYY format. Your application needs to validate the date, generate a due date 30 days into the future and ensure the due date is not a Sunday.

```
LET due_date = DTN(invoice_date$, "MMDDYY", ERR=BAD_DATE) + 30;
IF NTD(due_date, "Day") = "Sun"
    due_date =: + 1
```

When dealing with SQL dates, your program work variables should always be in storage format. The only time you will need to use another format is input and display/reporting. And both of these are accomplished with Thoroughbred *Basic* functions. Can your date routines be this simple?

```
Compare:      IF date_1 > date_2
Arithmetic:   LET date2 = date_1 + 30
Display:     PRINT NTD(date_1, date_mask$)
Validate input: LET date_1 = DTN(input_date$, date_mask$,
                                ERR=bad_date)
```

SYSTEM REQUIREMENTS

After settling on a format and methodology for your YEAR 2000 solution, you need to also consider potential walls you may run into. There are a number of problems outside of your control. These areas include O/S versions, hardware, serial devices, firmware, embedded micro code, other non-Thoroughbred software applications, and even Thoroughbred Environments. This is simply a domino effect. If the O/S cannot process YEAR 2000 dates, then Thoroughbred *Basic* cannot; and if Thoroughbred *Basic* cannot handle YEAR 2000 dates, then your application cannot.

Recent media coverage has done a fair job of exposing various computer components (hardware and software) not Y2K ready. For example, some PC BIOS will not properly rollover to the year 2000; instead the system date starts over with January 4, 1980. There are also some older UNIX/XENIX BIOS that also have problems recognizing the year 2000 and start over with 1900. You will need to do your own homework in this area. Change your operating system date to December 31, 1999 and your time to 11:58 PM. Shutdown your computer and wait a few minutes to reboot. Check your operating system date. Does it read January 1, 2000? Many of the newer machines are built to handle the century rollover. Are your installations using the latest hardware? If not, you are not alone. Many legacy installations continue to operate on what might be considered outdated hardware.

Once you get past your hardware/software operating environments, you need to evaluate your development platform (i.e. Thoroughbred). There are a couple of known Thoroughbred issues that you must consider and familiarize yourself with, before deciding on which version(s) of the Thoroughbred Environment will be used as a base for your Y2K solution.

First, Thoroughbred *Basics* prior to version 8.3.1 (released Oct. 1995) will either not start or return a bad value for DAY if the O/S system date is greater than December 31, 1999.

- T/OS Error=40 trying to SETDAY with a YEAR of 00
- Level 7 *Basic* the DAY variable is set to 01/01/:0 on January 1, 2000
- 8.20 through 8.30 Thoroughbred *Basic* gets an error trying to start after 12/31/1999

Second, Thoroughbred *Basics* prior to 8.4.0 will return the wrong century (50-year rule) when using the Thoroughbred *Basic* SQL date function DTN with 6-character dates when the system date is greater than December 31, 1999. For example, 8-character dates:

```
DTN("12311905","MMDDYYYY") returns 695788
DTN("12312005","MMDDYYYY") returns 732313
```

Now assume the current year is 2000. This 6-character example is wrong:

```
DTN("123105", "MMDDYY") returns 695788
```

The 50-year rule should determine the year is 2005 and return an SQL value of 732313. Thoroughbred *Basics* prior to 8.40 incorrectly return an SQL numeric value of 695788 representing the year 1905.

Those are the known problems with the older Thoroughbred versions. The anomalies described above may or may not affect your application in different ways. You will have to make the ultimate decision on which version satisfies your Y2K requirements. For instance, let's review *Solution-IV Accounting* and how the above issues affect different releases. In preparation for YEAR 2000, *Solution-IV* implemented SQL dates in version 8.30. The known problem list above says Thoroughbred *Basic* 8.3.0 will not start after the year 1999, so that is not a solution. The known problem list above also states that Thoroughbred *Basic* version 8.3.1 does not handle the 50-year properly using DTN. *Solution-IV* is dependent on the date processing provided by *Dictionary-IV*. In version 8.3.1, *Dictionary-IV* utilizes DTN with 6-character masks. Therefore, version 8.3.1 is also not a viable platform for *Solution-IV*. So, even though *Solution-IV* version 8.30 prepared itself for Y2K support, anomalies found in Thoroughbred *Basic* and *Dictionary-IV* prevent *Solution-IV* from any Y2K support until 8.40.

Does your application use SQL dates? Does your application rely on the results of DTN with 6-character date masks? Do you rely on *Dictionary-IV* to handle your date processing? What Thoroughbred *Basic* version is currently found in your install base? You must have a handle on these questions before declaring you have a Y2K solution. Though you may settle on an earlier version of Thoroughbred *Basic*, Thoroughbred will continue to strongly recommend version 8.4.1. It is most ideal for Y2K development since it includes a number of new Y2K specific tools.

TESTING AND OTHER ISSUES

YEAR 2000 testing is all about dates and how they affect your application. And while the repair of 2-digit date fields is relatively simple, making small changes to a lot of programs makes the testing phase a potentially elaborate process. You must ensure the Y2K specific modifications correct YEAR 2000 without disrupting or introducing new problems into applications that had previously worked. Regression testing should validate that your modifications continue to perform with dates prior to the year 2000. Industry experts estimate testing could range from 50% to 70% of your total Y2K project. You should also be prepared for testing and maintenance to continue well after the year 2000.

What is Y2K testing? Some may describe it as an evil necessity. After spending enormous resources to finally solve the YEAR 2000 problem, your application will provide no more capabilities than it did before the conversion process. This investment of resources will defer and overshadow real enhancement requirements. Testing is a necessity and you should not expect any short-term payback, short of continual operation for your company into the next century.

Y2K testing must take into consideration all components of an installation. Remember, just because your application can handle Y2K dates, don't assume your total solution is Y2K ready. Be sure to test your application paying special attention to how it interacts with the Thoroughbred Environment, UNIX, Windows, hardware, and other devices.

Currently, the Y2K certification process is self-governing. There is no external organization responsible for auditing your solution and program code changes. It is your responsibility to ensure your date functionality has been tested to work within, across and into the next century.

TESTING YOUR APPLICATION – FOLLOW THE DATA

So, how do you test your application for year 2000? Unfortunately, there's no silver bullet or magic pill that solves the Y2K dilemma or provides complete testing and quality assurance. Every application is unique. Some applications are more date sensitive than others. Testing your Y2K modifications demands a complete understanding of the application. Your intimate knowledge is key and should be utilized in building a Test Plan. Completing all the steps in your test plan should be instrumental in providing documentation required by many Y2K pessimists. Plan for changes. As you uncover areas with problems during the testing phase, you may need to revise your Test Plan.

Your test plan should include some broad objectives. For example, every date input in your application should be tested with valid and invalid dates. The dates should cover dates in 1998, 1999, 2000, and 2001. Two specific dates to test are 01-Jan-2000 and 29-Feb-2000. The first one catches any 123199 expiration date logic built into your business rules and the second one verifies a valid leap year date. Your test plan should identify how to enter these dates. Do you

support 4-character years in your input? If not, is the century based on the current year? While you are testing date inputs, also be aware of date display. Do dates display properly? If the year is displayed with 2 characters, is the century implicit?

If there is one key to testing Y2K applications, it is *follow the data*. This will lead to how dates are entered, displayed, manipulated, used to generate new dates, and handed off to other applications. The data is your clue. Follow it. Some things to look for in each step of testing include computations that may result in negative dates (i.e. 00 – 99); improper sorting and compares; incorrect calculations; screens and reports not formatted properly; and premature expiration dates (i.e. 12/31/1999). Following the data may cross over into other applications or modules. You need to be able to identify and test when another module or application starts to send or receive Y2K dates. Your Test Plan should identify which applications should be tested simultaneously.

Many applications should be able to identify with this *follow the data* example. A simple invoicing or billing module should be able to collect or input invoices, accept payments and provide sufficient reporting. Start with the invoice entry process that includes an invoice date. This process should also generate at least a due date. Is the due date calculation correct? Does it account for end of a month, year, and century? Does it account for the day 29-Feb-2000? We now have two dates to follow. The next action that is effected by these dates is the process of recording payments. If the invoice date is November 1999, can I have a partial payment in December 1999 and a second payment in January 2000? If the invoice date is December 1999 and the due date is January 2000, can payments be made in December 1999 and/or February 2000? Your Test Plan should identify the combinations of invoice, due, and payment dates required for testing. This simple example also allows testing of calculations, sorting and compares through reports. Our sample application provides Customer Aging reports. This may be in a form of Customer Inquiry, AR Trial Balance and/or Statements. Each of these reports will identify invoices past their due date using aging categories like 30, 60, and 90 days. Continue to follow the data. Invoices and payments may generate General Ledger Journal Entries. Does the General Ledger module support Y2K dates? Does the billing module send the date to General Ledger properly?

Testing will be an exhausting task. Entering YEAR 2000 dates into all of your date inputs is just the beginning. Potential showstopper problems will only be uncovered if you *follow the data*. Your Test Plan documents every procedure that is exercised during the test phase. The Test Plan will most likely be updated multiple times before you complete all of your testing.

OTHER ISSUES

Thoroughbred recognizes the YEAR 2000 transition is a serious business problem affecting all users throughout the software industry. For that reason, Thoroughbred does not minimize the task facing developers and is committed to assisting and providing solutions to meet the challenge. Thoroughbred's Consulting Services can support and assist during all phases of a YEAR 2000 project. Here are some helpful considerations that they have uncovered with some legacy applications:

- key words stored in date field – HOLD, ASAP
- expiration dates – if no expiration, set date to 12/31/99
- positional reads – READ(ch,KEY="9708",DOM=. . .)
- hard-coded century on Business Forms – do your checks have a pre-printed 19?
4-character dates in history files (YYMM)

During the data conversion phase, a simple way to test whether a date has been converted to an SQL date, is to test the first character equal to a "7." The SQL numbers 700000 through 799999 cover all dates between July 13, 1917 and April 26, 2191.

The YEAR 2000 project may seem overwhelming at times. Applications that require an interface to horizontal accounting modules may find it easier and less time consuming to update the accounting modules to something that already supports YEAR 2000. This may be a perfect time to consider *Solution-IV*. The *Solution-IV* Accounting modules are extremely robust and already provide YEAR 2000 support.

Finally, by assessing and planning to handle the conversion and upgrade workload now, the entire process should result in a less disruptive YEAR 2000 transition as well as higher customer satisfaction.