The Time For Dates

by Andy Forget

Since the very onset of the computer age, computer software and hardware has dealt with dates and times. Chronology has always been an integral component of our existence as practitioners of computer science. If we were to lose the ability to accurately record the dates and times of critical enterprise events, we would find ourselves technologically behind the ancient Sumerians, members of a civilization that invented writing more than six thousand years ago, to record critical transactions of their day: taxes, cattle, land, pottery, metalwork, and gold trading.

The development of the calendar was the first step towards accurately recording events. The actual solar year is 365 days 5 hours 48 minutes and 46 seconds, so when the ancients established a year as 365 days, they noticed a problem, Over time, January would slip into solar December, and every 4 years, the calendar would fall back approximately one day. After several centuries, if the problem was left unchecked, January would occur in the middle of summer in the northern hemisphere.

Because the year is not evenly divisible by a whole number of days, the practice arose of making arbitrary divisions and inserting extra days, or even months, into a year. Julius Caesar, with the help on Sosigenes, rectified the randomness of this situation by developing the Julian calendar in 63 B.C. The unique feature of the Julian calendar was that every fourth year was a leap year causing February to have 29 days. Due to advances in astronomy, by 1582, Pope Gregory XIII (Ugo Buoncompagni) ordained that 10 days be dropped and that years ending in hundreds be leap years if divisible by 400. This new twist on the Julian calendar resulted in the Gregorian calendar, adopted in England in 1752, and now used as the standard calendar today.

The Sumerians wrote on clay tablets. We write on silicon, magnetic and optical media. Both civilizations have been faced with information recording challenges. The Sumerian system for recording dates was adequate for their day. When properly implemented, the Gregorian calendar can work well for our business purposes (there are other equally important religious calendars). However, there is no general agreement on a contemporary method for recording chronological information. Many UNIX systems internally represent time as the number of seconds since January 1, 1970. The VMS system time mechanism is based upon the number of 100-nanosecond intervals since 00:00 hours, November 17, 1858 (the base time for the Smithsonian Institution astronomical calendar). The PRO/5 JUL() function produces a Julian number; the number of days since January 1, 4713 B.C.

"For tribal man space was the uncontrollable mystery.
For technological man it is time that occupies the same role."

MARSHALL MCLUHAN
The Mechanical Bride

Our Problem

In the Business BASIC world there are as many mechanisms to store a date as I have fingers on my hands. These mechanisms were developed by independent application developers to suit the needs of the day. The only problem, the day was in the year 1975. The system that many of us have embraced will fail on the first minute, of the first hour, of the twenty-first century!

Most of these dates were stored as strings, with substrings, that referred to the month, day, and year. They all fall into a variation of the YY/MM/DD format. These formats are used in millions of data files in thousands of applications. Let's look at just a few of these legacy date formats:

	YEAR	***MONTH***	***DAY***
MAI	*asc(c$(1,1))-129*	*asc(c$(2,1))-129*	*asc(c$(3,1))-129*
AON	*asc(c$(1,1))-32*	*asc(c$(2,1))-32*	*asc(c$(3,1))-32*
SOA	*asc(c$(1))-32*64+asc(c$(2,1))-32*	*num(c$(3,2))*	*num(c$(5,2))*
SSI	*num(c$(1,2))*	*num(c$(3,2))*	*num(c$(5,2))*

There are a few scary facts that appear right away:

1. None of these methods are easily interchangeable.
2. These date storage mechanisms will fail on or before the year 2000.
3. There is no way to easily calculate days in the future or past.
4. There is no easy way to test the validity of a date.

The dates are not interchangeable because there simply was no standard way to store a date when these mechanisms were developed, and so each storage mechanism was developed independently to meet the needs of the product at the time of development.

Regardless of the rationale behind the storage mechanism, storing the year as a two digit number is simply asking for trouble. An example of a current problem with the two digit programming comes from my grandmother, Pauline Forget. She was born in the year 1899 and when she goes to the Division of Motor Vehicles to renew her drivers license, their software reports that she has not been born yet.

One track used to overcome this problem is to check if the last two digits of the birth year were greater than the last two digits of the current year. If so, the application was programmed to assume that the person was born in the last century. This track will only work until the turn of the century; for even if we solve the two century dilemma, we run into a problem when we span three centuries. Again, my grandmother offers us a good example: if she was born in 1899, first received her driver's license in 1938, and has her first speeding ticket in 2003, current applications could not handle this information. Since mandatory euthanasia is not an acceptable social solution to this problem, we must find a technical one.

And what do we do when we sort on two digit dates that span a century? The years 1999, and 1997 sort correctly (1996, 1997, 1999) when we use the two digit year. As soon as we introduce a 21st century date, like 2003, we end up with the wrong sort (2003, 1996, 1997, 1999)!

Also, let's hope that nobody uses the year "00" as a marker of any kind. In less than 4 years your markers will be valid dates!

OK, one more terrifying scenario: what if you check the date of records for backup against the current date and destroy (or simply not backup) records older than a certain date. After the year 2000, all of your records will fail this criteria and you will suffer an absolute destruction of your data!

These examples make it clear that two digits are not enough to work with when we are dealing with dates. Not only do we have to change the physical format of our dates, we also have to change each application, procedure, and line of code that references dates. A medium sized accounting application with 5 modules will constitute more than 500 programs and over 60 data files. After you have changed your application data and programs to deal with 4 digit years, you will have to modify each entry screen and report to deal with 4 digit years! This simple formula will show you if you can solve the millennium problem before it is too late:

W = ((((P+D+S)/R)*U)+I)/40

W is the number of weeks it will take you to convert. P is the number of programs affected. D is the number of data files that need conversion. S is the number of screens that need modified. R is the number of maintenance programmers that are available to do the work. U is the number of hours required to modify a single unit. I is the number of hours of integration that will be required after all the maintenance work is done. Assuming we have 500 programs, 60 data files, 12 screens, and 3 maintenance programmers who can modify a program, data file, or screen in 5 hours, it will take us 2 weeks (80 hours) to integrate and the total time investment will be 25.8 weeks ((((500+60+12)/3*5)+80)/40. This does not include design, quality assurance, field trials, alpha testing, and beta testing.

We must address this problem today or we simply won't have time to solve it in the future!

Don't wait until the day before the millennium New Year to deal with this problem. Even if you are good enough to fix your programs in that small amount of time, January 1, 2000 is a Saturday. By the time your customers recover from the largest New Year's Eve party ever, the damage will have been done. Plan to be out of town.

If you are in town and plan to slip into the office on Saturday just to see how things are going, don't bother; your security card to the office will have expired because your security company uses a two digit year system.

Obviously, Business BASIC is not the only software that is challenged with this problem. A cottage industry has formed around fixing this problem in legacy COBOL systems. Many hardware platforms have intrinsic date problems including the Unisys 2200, IBM S390, and Intel based PC's. Yes, when DOS crosses the millennium you will suddenly be sent back in time to the year 1980. I hope you liked the Bee Gees.

A Solution

When we finally get the nerve to confront this dilemma we should choose a storage mechanism that is accurate, durable, and flexible.

To have an accurate mechanism, we have to properly deal with leap years, In addition to the year 2000 being the year that our "year odometers" will turn over, it is also a leap year (remember that Pope Gregory VIII decided that years ending in 00, and are divisible by 400, are leap years). Early releases of Excel, Lotus, and Quattro Pro do not treat February 29, 2000 correctly. Accuracy dictates that we should also be able to test the validity of a date: if an operator enters "June 31" as the delivery date for an order and the software accepts it, we have a problem.

A durable solution requires us to adopt a mechanism that will work well into the future. An IBM patch for the AS/400 simply puts the problem off until 2040. This just puts the problem off to another generation of programmers. On New Year's Day 2000, I do not want to be on a plane which is relying upon air traffic control software that has a millennium problem. Similarly, when I am 77 in 2040, I don't want to find out that all 50 of my grandchildren perished in an AS/400 software related accident when they were on the way to a surprise family reunion.

A flexible solution demands that we can easily perform several common operations on a date without having to write a whole lot of code. We should be able to add arbitrary days to a date and get a correct result. I should be able to add 7 days to today and get next

week. I should be able to easily look at a history record and know that it is older than 1095 days (3 years) and move in to an off-line archive. I should be able to easily ensure that our once a month backup occurs on a Friday. It should be easy to make sure that we promise no product shipments on Sunday.

The solution I propose has been available for more that 5 years in the PRO/5 (and BBx) JUL() and DATE() functions. The JUL() function takes a year, month, and day argument and produces a Julian number. The following table shows just a few things we can do with these versatile functions:

OPERATION	***DESCRIPTION***
jul(0,0,0)+7	*A week from today*
order_date+7	*A week from the order date*
if date(0:"%Ds")="Fri" then	*Is today Friday?*
if trans_date<jul(0,0,0)=1095 then	*Was the transaction more than three years ago?*

The first problem you might come across is converting a human readable date such as "10 Jan 1997" into a Julian number for storage. Conveniently, the PRO/5 extended utilities come equipped with a utility that performs just such a function. The utility **`_undate.utl`** takes as an argument the human readable form of the date, a mask describing that form, and returns the Julian value associated with the human readable string. The masks that **`undate`** accepts are the same as those accepted by the DATE() function:

FORMAT d		***z***	***s***	***l***
%Y	*1996*	*96*	*1996*	*1996*
%M	*3*	*03*	*Mar*	*March*
%D	*21*	*21*	*Tue*	*Tuesday*

Using the date function with a date format of "%D1 %M1 %Dd, %Yd" for the Julian value 2450115 would produce the string "Thursday February 1, 1996". Passing this string to the **`_undate`** utility, we can easily convert it back to the Julian number 2450115.

The next problem you face is storing this numeric value. There has never been an intrinsic Business BASIC date type. Traditional Business BASIC systems stored dates in either 3 or 6 byte terminated strings. The strings contained no binary information that could be interpreted as a terminator (often $0A$). If we represented the new Julian date as a binary integer, and if you have line-feed terminated fields in your record, the integer value could be construed as a terminator when it appears prior to any of the variable length fields. For example, the Julian date for April 12, 1996 is 2450186. The binary hex representation of this value is $0025630A$. It contains, by sheer coincidence, a line-feed character (the 4th byte of the binary string). If you were reading values form a data file using an I/O list, your Business BASIC would have problems distinguishing where fields began and ended.

If you use PRO/5 templates and *do not* use any variable length data, you can store Julian values as a 4 byte integer. Storing the Julian value as a numeric means that there will be no subsequent conversion required and operations against the date value will be more natural. I strongly recommend this method of date storage.

Another storage option simply stores the data as a 7 byte character string, or in Business BASIC, stores it as an ASCII numeric. For example, April 12,1996 would be stored as the string "2450186". This eliminates the binary data problem and is easy to perform operations against, but does cost 7 bytes of disk space for each date.

Part of the rational behind the original OEM date formats was the conservation of disk space, but in this day and age, saving disk space is the least of our problems. However, there is a certain amount of macho programmer mystic about saving a few bytes, and other legacy constraints associated with these existing systems. If you simply *must* get your dates to fit into 6 bytes, try this:

```
d$=hta(bin(julian_value,3))
```

This will result in a 6 byte string that must be converted in the following method after reading it from a file:

```
julian_value=dec(ath(d$))
```

This mechanism, known as the SSJ date format, is fine. It is durable and accurate, but does require a conversion step prior to being flexible.

The Rest of the Planet

While we have been mucking around with our own internal date formats, the world has not stood still. There is an ISO standard for storing and manipulating dates as well as an SQL standard. Unless we plan to integrate our applications with an ISO conformant application, we won't have to worry about the ISO standard, but we do have to worry about the SQL standard.

What we want to do is allow third-party applications the ability to browse our data. In particular, we would like ODBC compliant applications to use the BASIS ODBC Driver and add value to our existing systems. In the guise of the Microsoft ODBC specification, the ANSI x3.135 SQL standard calls for a uniform date format. The ODBC date construct is a structure with the following format:

```
type MY_DATE struct
  {
  year  SWORD;
  month UWORD;
  day   UWORD;
  };
```

Any date information coming from an ODBC driver to an application must adhere to this format. This means that all of the proprietary

formats must be converted to ODBC format in order for the ODBC compliant application to understand them. There is nothing more satisfying than looking at an Excel spreadsheet and seeing your date displayed in a human readable format.

Fortunately, the BASIS ODBC Driver can recognize certain native date formats. By default, if a numeric field (BBx types N, I, U, X, Y, B, and D) has a column name that ends in "DATE:," like SHIP_DATE, the BASIS ODBC Driver will assume that the numeric value is Julian and do an automatic conversion into the ODBC date format. If you adopt the conversion of storing your dates as numeric Julians (for example. ASCII numeric or integer) you will automatically benefit from this conversion. *[In version 2.0 of the ODBC Drive, a date extension must be explicitly specified in the ODBC configuration dialog. In addition, Leaving the OEM Type option unset in the configuration dialog will cause the ODBC Driver to evaluate columns as Julians.]*

Additionally, the BASIS ODBC Driver can recognize certain OEM formats. If you modify your odbc.ini file and add the line **`DATETYPE=xxx`** (where xxx is either MAI, SOA, AON, SSI, or SSJ) to your datasource, the BASIS ODBC Driver will treat *character* fields which have a column name that ends in "DATE" as the respective OEM date format. *[With the BASIS ODBC Driver 2.0, which does not use an odbc.ini file, this option should be set using the ODBC Datasource Administrator. This can be accessed through the ODBC icon in the Win95/NT control panel.]*

Because of the support for OEM dates provided by the BASIS ODBC Driver, it is possible to create an Access or Visual Basic application that can not only *read* and understand the OEM date formats, but will be able to *insert* and *update* new and existing date values into existing data files.

Sadly, except for the SSJ format, all of the OEM date formats supported by the BASIS ODBC Driver will assume all years are in the 20th century (the 1900's) because they only provide two-digit year fields.

Conclusion

Perhaps it wasn't the Babylonians who conquered the Sumerians. Maybe it was a lack of a consistent and flexible mechanism to record dates which eventually spelled the end for the Sumerian civilization. If you don't want to have a hand in the fall of Western Civilization, you should start thinking about your millennium date problem now!

A great place to get further information about the year 2000 problem is on the Year 2000 Web site http://www.year2000.com/. I just surfed this site and the first thing Year 2000 told me was that I had 3 years, 333 days, 8 hours, 51 minutes, and 44 seconds to get my act together.