IDRBT Working Paper No. 9

Y2K38

- *Ashutosh Saxena and Sanjay Rawat*

ABSTRACT

Y2K problems were born because programs did not allocate enough digits to represent that year. Y2K38 problems arise out of programs not allocating enough bits to internal time. It is another date problem, which results from computing dates moving into the year 2038 and beyond in 32-bit operating systems. Unix and other C applications represent time as the number of seconds from January 01, 1970. The 32-bit variable `time_t` *that stores this number overflows in the year 2038 and becomes Friday, December 13, 1901. The precise date and time of this occurrence is Tuesday, Jan 19, 03:14:07, 2038. Even today, any date calculations forecasted beyond this time will be erroneous.*

Banks, Financial Institutions and Insurance Companies, which have servers running on Unix or Unix like OS, will need to react now, as many product and services offered by them are for longer duration such as Loans, Bonds and Polices, etc. Modifications in the source code of the current applications, as given in the paper and/or switching to 64-bit computing is required to solve the problem.

1.0 Introduction

Modern Unix time is based strictly on Coordinated Universal Time (UTC). The UTC counts time using SI seconds, and breaks up the span of time into days. The UTC days are mostly 86,400 seconds long, but are occasionally of 86,401 seconds or 86,399 seconds whenever the days do not synchronise with the rotation of the Earth.

The Unix epoch is the time 00:00:00 UTC on January 01, 1970. The Unix time number is zero at the Unix epoch, and increases by exactly (60*60*24 =) 86,400 seconds per day since the epoch. Thus 2004-09-16T 00:00:00, 12,677 days after the epoch, is represented by the Unix time number 12677 * 86400 = 1095292800. Within each day, the Unix time number is calculated at midnight UTC (00:00:00), and increases by exactly 1 per second since midnight. Thus, 2004-09-16T17:55:43, (17*60*60+55*60+43 =) 64,543 seconds since midnight on the day in the example, is represented by the Unix time number: 1095292800 + 64543 = 1095357343.

This means that on a normal UTC day of duration of 86,400 seconds, the Unix time number changes in a continuous manner across midnight. For example, at the end of the day used in the examples above, the time representations progress as below:

UTC	Unix time
2004-09-16T23:59:58	1095379198
2004-09-16T23:59:59	1095379199
2004-09-16T23:59:59	1095379199
2004-09-16T23:59:59	1095379199
2004-09-16T23:59:59	1095379199
2004-09-17T00:00:00	1095379200
2004-09-17T00:00:00	1095379200
2004-09-17T00:00:00	1095379200
2004-09-17T00:00:00	1095379200
2004-09-17T00:00:01	1095379201
2004-09-17T00:00:01	1095379201

A Unix time number, as shown above, can be represented in any form capable of representing numbers. In some applications, the number is simply represented textually as a string of decimal digits, raising only trivial additional issues. However, there are certain binary representations of Unix times that are of particular significance.

The standard Unix time_t (data type representing a point in time) is a signed integer data type, of 32-bits or more, directly encoding the Unix time number. Being integer means that it has a resolution of one second; many Unix applications, therefore, handle time only to that resolution and being 32-bits, means it covers a range of about 136 years in total. The minimum representable time is 1901-12-13T20:45:52 and the maximum representable time is 2038-01-19T03:14:07. Therefore at 2038-01-19T03:14:08, this representation will overflow.

1.1 The Problem

It is quite possible that in the first month of the year 2038 many computers will encounter a date-related bug resulting in inaccurate dates. Just as Y2K problems arose from programs not allocating enough digits to that year, Y2K38 problems would arise from programs not allocating enough bits to internal time.

The effect of this bug is hard to predict. Some experts are of the opinion that this bug can cause serious problems on many platforms, especially Unix and Unix-like platforms, because these systems will “run out of time”. Starting at GMT 03:14:07, Tuesday, January 19, 2038, one can expect many systems around the world to break down, satellites to fall out of orbit, massive power outages (like the 2003 North American blackout), failures of life support systems, interruptions of phone systems, banking errors, etc. One second after this critical second, many of these systems will have wildly inaccurate date settings, producing all kinds of unpredictable

consequences. In short, many of the awful predictions for the year 2000 are more likely to actually occur in the year 2038. One may consider the year 2000 as just a dry run.

Most of the Banks, Financial Institutions and Insurance Companies whose servers run on Unix or Unix-like OS, will need to react now, as many product and services offered by them such as Loans, Bonds and Polices, etc., are for a longer duration. This problem of insufficient storage for date variable can cause major errors in their books of account and can result in a major disaster. Software for mortgage calculation, vital statistics, and many other applications may need to frequently and reliably examine a century or two forward and backward.

The problem results from computing dates moving into the year 2038 and beyond in 32-bit operating systems. Unix and other C (and possibly some JAVA) applications represent time as the number of seconds from January 01, 1970. The 32-bit variable (time_t) that stores this number overflows in the year 2038 and becomes December 13, 1901. However, even today, any date calculations forecasted beyond that time will be erroneous.

By the year 2038, the time_t representation for the current time will be over 2,140,000,000 and that is the problem. A modern 32-bit computer stores a "signed integer" data type, such as time_t, in 32-bits. The first of these bits is used for the positive/negative sign of the integer, while the remaining 31-bits are used to store the number itself. The highest number these 31-data bits can store works out to exactly 2,147,483,647. A time_t value of this exact number, 2,147,483,647, represents Tuesday, January 19, 2038, at 7 seconds past 3:14 A.M. Greenwich Mean Time. So, at 3:14:07 AM GMT on that fateful day, every time_t used in a 32-bit C or C++ or JAVA program will reach its upper limit. One second later, on 19 January 2038 at 3:14:08 AM GMT, disaster strikes. Table 1. gives more clarity on the exact time_t representations.

Table 1. Exact time_t *representations of Date and Time in signed 32-bits datatype.*

Date & time	time_t **representation**
1-Jan-1970, 12:00:00 AM GMT	0
1-Jan-1970, 12:00:01 AM GMT	1
1-Jan-1970, 12:01:00 AM GMT	60
1-Jan-1970, 01:00:00 AM GMT	3 600
2-Jan-1970, 12:00:00 AM GMT	86 400
3-Jan-1970, 12:00:00 AM GMT	172 800
1-Feb-1970, 12:00:00 AM GMT	2 678 400
1-Mar-1970, 12:00:00 AM GMT	5 097 600
1-Jan-1971, 12:00:00 AM GMT	31 536 000
1-Jan-1972, 12:00:00 AM GMT	63 072 000
1-Jan-2003, 12:00:00 AM GMT	1 041 379 200
1-Jan-2038, 12:00:00 AM GMT	2 145 916 800
19-Jan-2038, 03:14:07 AM GMT	2 147 483 647

1.2 The Reason

Unix and Unix-like operating systems do not calculate time according to the Gregorian calendar. They simply count time in seconds since their arbitrary “birthday”, GMT 00:00:00, Thursday, January 01, 1970. The industry-wide practice is to use a 32-bit variable for this number (32-bit signed time_t). Imagine an odometer with 32-wheels, each marked to count from 0 and 1 (for base-2 counting), with the end wheel used to indicate a positive or negative integer.

The largest possible value for this integer is 2**31-1 = 2,147,483,647 (over two billion). The 2,147,483,647 seconds after Unix's birthday corresponds to GMT 03:14:07, Tuesday, January 19, 2038. One second later, many Unix systems will revert to their date of birth (like an odometer rollover from 999999 to 000000). Since the end bit indicating positive/negative integer may flip over, systems may revert the date to 20:45:52, Friday, December 13, 1901 (which corresponds to GMT 00:00:00 Thursday, January 01, 1970 minus 2**31 seconds).

1.3 The Impact

So far, the few operating systems that are not found susceptible to the 2038 bug include very new versions of Unix and Linux ported to 64-bit platforms (refer online list at the site in [1]). A large number of machines, platforms, and applications could be affected by the 2038 problem. Most of these will, hopefully get decommissioned before the critical date. However, it is possible that some machines in service now, or legacy systems, which have never been upgraded due to budget constraints, may still be operating in 2038. These may include process control computers, space probe computers, embedded systems in traffic light controllers, navigation systems, etc., and it may not be possible to upgrade many of these systems.

The clock circuit hardware that has adopted the Unix time convention may also be affected if 32-bit registers are used. While 32-bit CPUs may become obsolete in desktop computers and servers by 2038, they may still exist in micro-controllers and embedded circuits. Embedded functions present a serious maintenance problem for all roll-over issues like the year 2038 problem, since the package part number and other markings usually give no indication of the device’s internal function. For instance, the Z80 processor was available till late 2000 as an embedded function within programmable devices. Such embedded functions present a serious maintenance problem for Y2K38 and similar rollover issues.

This problem is somewhat easier to fix than the Y2K problem on mainframes. Well-written programs can simply be recompiled with a new version of the library that uses, for example, 8-byte values for the storage format. This is possible because the library encapsulates the whole time activity with its own time types and functions.

Some Unix vendors have already started using a 64-bit signed time_t in their operating systems to count the number of seconds since GMT 00:00:00, Thursday, January 01, 1970. Programs or databases with a fixed field width should probably allocate at least 48-bits to storing time values. A 64-bit Unix time would be safe for the indefinite future, as this variable would not overflow until 2**63 or 9,223,372,036,854,775,808 (over nine quintillion) seconds after the beginning of the Unix epoch - corresponding to GMT 15:30:08, Sunday, 4th December, 292,277,026,596.

This is a rather artificial and arbitrary date, considering that it is several times the average lifespan of the Sun, the very same celestial body by which we measure time. The Sun is estimated at present to be about four-and-a-half billion years old, and it may last another five billion years before running out of hydrogen and turning into a white-dwarfed star. Thus a straightforward approach is to switch to 64-bit computing and solve the problem.

2.0 Test It Out Yourself

If you are a programmer or a systems integrator, who needs to know what you can do to solve this problem, a quick check with the following Perl and C script may help in determining if your computers are susceptible to the problem (this requires Perl and/or C to be installed on your system):

Perl

```
$ENV{'TZ'} = "GMT";
for ($clock = 2147483641; $clock <
2147483651; $clock++) print ctime($clock);
```

C

```
#include <stdio.h>
#include <time.h>
int main()
{
 struct tm *tm_ptr;
 time_t i;
 i=2147483641;
 for(;i<2147483651;i++){
      gmtime_r(&i,tm_ptr);
printf("Date: %04d-%02d-%02d ",tm_ptr-
>tm_year+1900, tm_ptr->tm_mon+1, tm_ptr-
>tm_mday);
printf("Time: %02d:%02d:%02d\n", tm_ptr->tm_hour,
tm_ptr->tm_min, tm_ptr->tm_sec);
 }
  exit(0);
}
```

The output of this script when executed in our lab on GNU/Linux (kernel 2.4.20-8) for both

```
Tue Jan 19 03:14:01 2038
Tue Jan 19 03:14:02 2038
Tue Jan 19 03:14:03 2038
Tue Jan 19 03:14:04 2038
Tue Jan 19 03:14:05 2038
```

```
Date: 2038-01-19 Time: 03:14:01
Date: 2038-01-19 Time: 03:14:02
Date: 2038-01-19 Time: 03:14:03
Date: 2038-01-19 Time: 03:14:04
Date: 2038-01-19 Time: 03:14:05
```

Tue Jan 19 03:14:06 2038	Date: 2038-01-19 Time: 03:14:06
Tue Jan 19 03:14:07 2038	Date: 2038-01-19 Time: 03:14:07
Fri Dec 13 20:45:52 1901	Date: 1901-12-13 Time: 20:45:52
Fri Dec 13 20:45:52 1901	Date: 1901-12-13 Time: 20:45:53
Fri Dec 13 20:45:52 1901	Date: 1901-12-13 Time: 20:45:54

The last three outputs are not as expected. You may check whether the date and time outputs are correct after the critical event second in your machine?

2.1 A Checklist

1. An organisation called The Open Group (formerly X/Open), which maintains the Unix specification and trademark, has a number of programming recommendations [2] that should be followed by developers to deal with the year 2038 and other problematic dates.
2. If you are working with Open Source code, a free library at [3] may be a useful reference for patching existing code for high-accuracy long-term time calculation
3. Check out an article [4] regarding Solutions to the Year 2000 Problem. Though fixing the Y2K bug may not directly help in fixing the Y2K38 bug but many of the suggestions from this article can be applied to the 2038 problem too.

2.2 How about making time_t unsigned in 32-bit software?

A solution that has been suggested for the existing 32-bit software is to re-define time_t as an unsigned integer instead of a signed integer. It does sound like a good idea at first. We already know that most of the standard time_t handling functions do not accept negative time_t values anyway, so why not just make time_t into a data type that only represents positive numbers?

An unsigned integer does not have to waste one of its bits to store the plus/minus sign for the number it represents. This doubles the range of numbers it can store. Whereas a signed 32-bit integer can only go up to 2,147,483, 647, an unsigned 32-bit integer can go all the way up to 4,294 967,295. A value of time_t of this magnitude could represent any date and time from 12:00:00 AM 01, Jan, 1970 all the way to 6:28:15 AM 7-Feb-2106, which surely gives us more than enough years for software to dominate the planet.

Well, there is a problem time_t, which is not just used to store absolute dates and times. It is also used, in many applications, to store differences between two date/time values, i.e. to answer the question of “how much time is there between date A and date B?” In such cases, we do need time_t to allow negative values. It is entirely possible that date B comes before date A. Blindly changing time_t to an unsigned integer will, in these parts of a program, make the code unusable.

Changing time_t to an unsigned integer would, in most programs, be like borrowing a second debt to repay the first debt. One would be fixing one set of bugs (the Year 2038 Problem) only to introduce a whole new set of bugs, as the time differences, are not being computed properly.

2.3 Not very obvious

The greatest danger with the Year 2038 problem is its invisibility. The more-famous Year 2000 drew great attention because it only takes a few seconds of thought, even for a computer-illiterate person, to imagine what might happen when 1999 turns into 2000. This change also received huge media attention. But January 19, 2038 is not as obvious.
Software companies will probably not think of trying out a Year 2038 scenario before doomsday strikes. Of course, there will be some warning ahead of time. Scheduling software, billing programs, personal reminder calendars, and other such pieces of code that set dates in the near future will fail as soon as one of their target dates exceeds 19-Jan-2038, assuming a time_t type variable is used to store them.

Worse, the parts of their software, which they had to fix for Year 2000 compliance, will be completely different from the parts of their programs that will fail on 19, Jan 2038, so fixing one problem will not fix the other. Most programs written in the C or other high level programming language were relatively immune to the Y2K problem, but may suffer instead from the Year 2038 problem. This problem arises because most C programs use a library of routines called the standard time library (time.h). This library establishes a standard 4-byte format for the storage of time values, and also provides a number of functions for converting, displaying and calculating time values.

3.0 The Other Side

One school of thought feels that this impending disaster will NOT strike too many people and thus do not consider it a problem, reasoning that, by the time 2038 rolls around, most programs will be running on 64-bit or even 128-bit computers. In a 64-bit program, a time_t could represent any date and time in the future till 292,000,000,000 A.D., which is several times the currently estimated age of the universe.

The problem with this kind of optimism is the same root problem behind most of the Y2K concerns that plagued the software industry Legacy Code. Developing a new piece of software is an expensive and time-consuming process. It is much easier to take an existing program that we know, and code one or two new features into it, than to throw the earlier program out and

write a new one from scratch. This process of enhancing and maintaining “legacy” source code can go on for years, or even decades.

The MS-DOS layer, still at the heart of Microsoft's Windows 98 and Windows ME was first written in 1981. Many of the financial software hit by the various Y2K bugs were also being used and maintained since the 1970s, when the year 2000 was still thought of as more of a science fiction movie than the actual impending future. Surely, if those software had been written in the 1990s, their Y2K compliance would have been crucial to its authors, and those software would have been designed with the year 2000 in mind.

One may note that computer designers can no longer afford to make a “clean break” with the computer architectures of the past. No one wants to buy a new kind of PC if it does not run all their old PC’s programs. So, just as the new generation of Microsoft Windows operating systems must be able to run the old 16-bit programs written for Windows 3 or MS-DOS, similarly any new PC architecture should be able to run existing 32-bit programs in some kind of “backward compatibility” mode. Even if every PC in the year 2038 has a 64-bit CPU, there will be a lot of older 32-bit programs running on them.

The other school of thought feels that the Y2K38 threat is more likely to result in aircraft falling from the sky, glitches in life-support systems, and nuclear power plant meltdown, than the threats imposed by Y2K problem, which was more likely to disrupt inventory control, credit card payments, pension plans, etc. They reason that the Y2K38 problem involves the basic system time-keeping from which most other time and date information is derived, while the Y2K problem, mostly involved application programs.

But it is necessary that we do our homework and keep ourselves updated properly in order to run our system accurately for the future.

3.1 The Possible Remedy

It is now clear that any solution to Y2K38 problem is neither trivial nor obvious. Every solution and suggestion has its pros and cons and in most of the cases, it is not feasible too. The most robust solution, which can be thought of now, seems to be bringing in the 64-bit OS. But as already mentioned, there are many applications, which need information of many years ahead of now, like loans, bonds, etc. For such applications, we provide here some solution that can minimise or possibly eliminate the risk of erroneous calculations. In the following paragraphs, we try to explain a possible solution to tackle Y2K38 problem.

The proposed solution makes use of a header file, provided by Paul Sheer [5], which is to be included in the source code of the application followed by recompilations to get the executable. However, this solution is applicable to only those applications whose source code is available in hand. After these modifications, the application software is Y2K38 compliant.

The solution is intended for the codes, which make use of the time function gmtime() and gmtime_r(). The gmtime() and gmtime_r() functions convert a time in seconds since the Epoch (00:00:00 UTC, January 01, 1970) into a broken-down time, expressed as Coordinated Universal Time (UTC). These function definitions are given as follows:

```
#include <time.h>
struct tm *gmtime(const time_t *clock); //where clock is the time to be converted.
```

```
#include <time.h>
struct tm *gmtime_r(const time_t *clock, struct tm *result);
```

where *clock* is the time to be converted and *result* points to the structure where the converted time is to be stored.

Any piece of code that makes use of above functions will fall prey on 19-Jan-2038, 03:14:07 AM GMT. But the danger can be avoided, if we replace the above functions, with the function defined, in the header file mentioned above. The prototype of the new function, called as pivotal_gmtime_r(), is as follows:

```
struct tm *pivotal_gmtime_r (const time_t *now, const time_t *future, struct tm *result);
```

where now is the current time, which can be obtained by calling time(&now) function, future is any time in future (beyond 2038) and result is the pointer to the structure, where the converted time is to be stored.

We have seen that maximum value of time_t variable is 2,147,483,647 i.e. 19-Jan-2038, 03:14:07 AM GMT. Therefore any program, which makes use of it should yield an unexpected date at time_t := 2 147 483 648. Now if we can show that by using the above defined function, the date beyond 19-Jan-2038, 03:14:07 AM GMT i.e. date for which time_t: = 2 147 483 648 or greater than this, the problem of Y2K38 can be solved for some specific codes. The following is the example of the code, which does the same.

```
#include <stdio.h>
#include <time.h>
#include "pivotal_gmtime_r_c.h"
int main()
{
        struct tm *tm_ptr;
        time_t now, beyond2038;
        time(&now);
        beyond2038=2147483641;
        for(;beyond2038<2147483651;beyond2038++){
```

```
        pivotal_gmtime_r(&now,&beyond2038,tm_ptr);
        printf("Date: %04d-%02d-%02d ",tm_ptr->tm_year+1900,
             tm_ptr->tm_mon+1, tm_ptr->tm_mday);
        printf("Time: %02d:%02d:%02d\n",tm_ptr->tm_hour,
            tm_ptr->tm_min, tm_ptr->tm_sec);
             }
             exit(0);
}
```

It is clear from the code that the last 3 iterations for the variable beyond2038 should produce some unexpected date (i.e. some date in year 1901), if the code is not Y2K38 compliant. Otherwise the output of the code should give the correct date i.e. 19 Jan2038, 03:14:08 AM GMT and so on. The following is the output of the program on Linux 2.4.20-8.

```
------------------------------------------------------------
Date: 2038-01-19 Time: 03:14:01
Date: 2038-01-19 Time: 03:14:02
Date: 2038-01-19 Time: 03:14:03
Date: 2038-01-19 Time: 03:14:04
Date: 2038-01-19 Time: 03:14:05
Date: 2038-01-19 Time: 03:14:06
Date: 2038-01-19 Time: 03:14:07
Date: 2038-01-19 Time: 03:14:08
Date: 2038-01-19 Time: 03:14:09
Date: 2038-01-19 Time: 03:14:10
------------------------------------------------------------
```

It is clear that the new function has made the code Y2K38 compliant. The date is calculated correctly even after the current limit of the variable time_t.

4.0 Concluding Remarks

The Y2K problem kept us engrossed and worried in the year 1999. However, the Y2K38 problem is not as obvious. This problem arises out of programs not allocating enough bits to internal time representation, which results from computing dates moving into the year 2038 and beyond in 32-bit operating systems. Even today, any date calculations forecasted beyond *Tuesday, Jan 19, 03:14:07, 2038,* will be erroneous.

Most of the Banks, Financial Institutions and Insurance Companies, which have servers running on Unix or Unix like OS, will need to react now, as many product and services offered by them are for longer duration such as Loans, Bonds and Polices, etc. Modifications in the source code of the current applications, as given here and/or switching to 64-bit computing is required to solve the problem.

Acknowledgements

Shri R. Gandhi, In-charge Director, IDRBT, for initiation and motivation. The Computing and Internet community working in this area too have helped in evolving this paper.

Appendix - I

pivotal_gmtime_r_c.h

```
/* pivotal_gmtime_r - a replacement for gmtime/localtime/mktime that works around the 2038 bug on 32-bit
                      systems. (Version 3)

Copyright (C) 2005  Paul Sheer

Redistribution and use in source form, with or without modification, is permitted provided that the above copyright
notice, this list of  conditions, the following disclaimer, and the following char array are retained.  Redistribution and use in binary form must reproduce an  acknowledgment: 'With software provided by http://2038bug.com/' in the documentation and/or other materials provided with the distribution, and wherever such acknowledgments are usually  accessible in Your program.

This software is provided "AS IS" and WITHOUT WARRANTY, either express or implied, including, without limitation, the warranties of NON-INFRINGEMENT, MERCHANTABILITY or FITNESS FOR A PARTICULAR

PURPOSE. THE ENTIRE RISK AS TO THE QUALITY OF THIS SOFTWARE IS WITH YOU. Under no circumstances and under no legal theory, whether in tort (including negligence), contract, or otherwise, shall the  copyright owners be liable for any direct, indirect, special, incidental, or consequential damages of any character arising as a result of the use of this software including, without limitation, damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses. This limitation of liability shall not apply to liability for death or personal injury resulting from copyright owners' negligence to the extent applicable law prohibits such limitation. Some jurisdictions do not allow the exclusion or limitation of incidental or consequential damages, so this exclusion and limitation may not apply to you.
*/

const char pivotal_gmtime_r_stamp[] =
   "pivotal_gmtime_r. Copyright (C) 2005  Paul Sheer. Terms and "
   "conditions apply. Visit http://2038bug.com/ for more info.";

/*
On 32-bit machines, with 'now' passed as NULL, pivotal_gmtime_r() gives the same result as gmtime() (i.e. gmtime() of the GNU C library 2.2) for all values of time_t positive and negative. See the gmtime() man page for more info.

It is intended that you pass 'now' as the current time (as previously retrieved with a call such as time(&now);). In this case, pivotal_gmtime_r() returns the correct broken down time in the range of now - 2147483648 seconds through to now + 2147483647 seconds

For example, on 10-Jan-2008, pivotal_gmtime_r() will return the correct broken down time format from 23-Dec-1939 through 29-Jan-2076.

For all values of 'now' before Jan-23-2005 and after Jan-19-2038, pivotal_gmtime_r() will return the correct broken down time format from exactly 01-Jan-1970 through to 07-Feb-2106.
```

In other words, if, for example, pivotal_gmtime_r() is used in a program that needs to convert time values of 25 years into the future and 68 years in the past, the program will operate as expected until the year 2106-25=2081. This will be true even on 32-bit systems.

Note that "Jan-23-2005" is the date of the authoring of this code. Programmers who have available to them 64-bit time values as a 'long long' type can use gmtime64_r() instead, which correctly converts the time even on 32-bit systems. Whether you have 64-bit time values will depend on the operating system.

Both functions are 64-bit clean and should work as expected on 64-bit systems. They have not yet been tested on 64-bit systems however.
The localtime() equivalent functions do both a POSIX localtime_r()and gmtime_r() and work out the time zone offset from their difference. This is inefficient but gives the correct timezone offset and daylight savings time adjustments.

The function prototypes are:

```
long long pivot_time_t (const time_t * now, long long *t);
long long mktime64 (struct tm *t);
struct tm *localtime64_r (const long long *t, struct tm *p);
struct tm *pivotal_localtime_r (const time_t * now, const time_t * t, struct tm *p);
struct tm *gmtime64_r (const long long *t, struct tm *p);
struct tm *pivotal_gmtime_r (const time_t * now, const time_t * t, struct tm *p);
```

pivot_time_t() takes a 64-bit time that may have had its top 32-bits set to zero, and adjusts it so that it is in the range explained above. You can use pivot_time_t() to convert any time that may be incorrect. pivot_time_t() returns its argument unchanged if either now is NULL or sizeof(time_t) is not 4.

mktime64() is a 64-bit equivalent of mktime().

localtime64_r() is a 64-bit equivalent of localtime_r().

pivotal_localtime_r() is 32-bit equivalent of localtime_r() with pivoting.

gmtime64_r() is a 64-bit equivalent of gmtime_r().

pivotal_gmtime_r() is a 32-bit equivalent of gmtime_r() with pivoting.

Note that none of these functions handle leap seconds.

RATIONALE: The purpose of pivotal_gmtime_r() is as a replacement for the functions gmtime(), localtime() and their corresponding reentrant versions gmtime_r() and localtime_r().

pivotal_gmtime_r() is meant for 32-bit systems that must still correctly convert 32-bit time into broken down time format through the year 2038. Since most programs tend to operate within a range of time no more than 68 years in the future or the past, it is possible to determine the correct interpretation of a 32-bit time value in spite of the wrap that occurs in the year 2038.

Many databases are likely to store time in 32-bit format and not be easily upgradeable to 64-bit. By using pivot_time_t(), these time values can be correctly used.

Changes:

```
  06-Feb-2005  v3.0:  Some optimizations.
                                    mktime() no-longer zeros tm struct.
*/
#include <stdlib.h>
```

```
#include <stdio.h>
#include <time.h>

static const int days[4][13] = {
   {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
   {31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
   {0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365},
   {0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366},
};

#define LEAP_CHECK(n)  ((!(((n) + 1900) % 400) || (!(((n) + 1900) % 4) && (((n) + 1900) % 100))) != 0)
#define WRAP(a,b,m)      ((a) = ((a) <  0  ) ? ((b)--, (a) + (m)) : (a))

long long pivot_time_t (const time_t * now, long long *_t)
{
   long long t;
   t = *_t;
   if (now && sizeof (time_t) == 4) {
      time_t _now;
      _now = *now;
      if (_now < 1106500000 /* Jan 23 2005 - date of writing */ )
         _now = 2147483647;
      if ((long long) t + ((long long) 1 << 31) < (long long) _now)
         t += (long long) 1 << 32;
   }
   return t;
}

static struct tm *_gmtime64_r (const time_t * now, long long *_t, struct tm *p)
{
   int v_tm_sec, v_tm_min, v_tm_hour, v_tm_mon, v_tm_wday, v_tm_tday;
   int leap;
   long long t;
   long m;
   t = pivot_time_t (now, _t);
   v_tm_sec = ((long long) t % (long long) 60);
   t /= 60;
   v_tm_min = ((long long) t % (long long) 60);
   t /= 60;
   v_tm_hour = ((long long) t % (long long) 24);
   t /= 24;
   v_tm_tday = t;
   WRAP (v_tm_sec, v_tm_min, 60);
   WRAP (v_tm_min, v_tm_hour, 60);
   WRAP (v_tm_hour, v_tm_tday, 24);
   if ((v_tm_wday = (v_tm_tday + 4) % 7) < 0)
      v_tm_wday += 7;
   m = (long) v_tm_tday;
   if (m >= 0) {
      p->tm_year = 70;
      leap = LEAP_CHECK (p->tm_year);
      while (m >= (long) days[leap + 2][12]) {
         m -= (long) days[leap + 2][12];
         p->tm_year++;
         leap = LEAP_CHECK (p->tm_year);
      }
      v_tm_mon = 0;
      while (m >= (long) days[leap][v_tm_mon]) {
         m -= (long) days[leap][v_tm_mon];
```

```
            v_tm_mon++;
        }
    } else {
        p->tm_year = 69;
        leap = LEAP_CHECK (p->tm_year);
        while (m < (long) -days[leap + 2][12]) {
            m += (long) days[leap + 2][12];
            p->tm_year--;
            leap = LEAP_CHECK (p->tm_year);
        }
        v_tm_mon = 11;
        while (m < (long) -days[leap][v_tm_mon]) {
            m += (long) days[leap][v_tm_mon];
            v_tm_mon--;
        }
        m += (long) days[leap][v_tm_mon];
    }
    p->tm_mday = (int) m + 1;
    p->tm_yday = days[leap + 2][v_tm_mon] + m;
    p->tm_sec = v_tm_sec, p->tm_min = v_tm_min, p->tm_hour = v_tm_hour,
        p->tm_mon = v_tm_mon, p->tm_wday = v_tm_wday;
    return p;
}

struct tm *gmtime64_r (const long long *_t, struct tm *p)
{
    long long t;
    t = *_t;
    return _gmtime64_r (NULL, &t, p);
}

struct tm *pivotal_gmtime_r (const time_t * now, const time_t * _t, struct tm *p)
{
    long long t;
    t = *_t;
    return _gmtime64_r (now, &t, p);
}

long long mktime64 (struct tm *t)
{
    int i, y;
    long day = 0;
    long long r;
    if (t->tm_year < 70) {
        y = 69;
        do {
            day -= 365 + LEAP_CHECK (y);
            y--;
        } while (y >= t->tm_year);
    } else {
        y = 70;
        while (y < t->tm_year) {
            day += 365 + LEAP_CHECK (y);
            y++;
        }
    }
    for (i = 0; i < t->tm_mon; i++)
        day += days[LEAP_CHECK (t->tm_year)][i];
    day += t->tm_mday - 1;
```

```
   t->tm_wday = (int) ((day + 4) % 7);
   r = (long long) day *86400;
   r += t->tm_hour * 3600;
   r += t->tm_min * 60;
   r += t->tm_sec;
   return r;
}

static struct tm *_localtime64_r (const time_t * now, long long *_t, struct tm *p)
{
   long long tl;
   time_t t;
   struct tm tm, tm_localtime, tm_gmtime;
   _gmtime64_r (now, _t, &tm);
   if (tm.tm_year > (2037 - 1900))
      tm.tm_year = 2037 - 1900;
   t = mktime64 (&tm);
   localtime_r (&t, &tm_localtime);
   gmtime_r (&t, &tm_gmtime);
   tl = *_t;
   tl += (mktime64 (&tm_localtime) - mktime64 (&tm_gmtime));
   _gmtime64_r (now, &tl, p);
   p->tm_isdst = tm_localtime.tm_isdst;
   return p;
}

struct tm *pivotal_localtime_r (const time_t * now, const time_t * _t, struct tm *p)
{
   long long tl;
   tl = *_t;
   return _localtime64_r (now, &tl, p);
}

struct tm *localtime64_r (const long long *_t, struct tm *p)
{
   long long tl;
   tl = *_t;
   return _localtime64_r (NULL, &tl, p);
}
```

References:

1. http://www.deepsky.com/~merovech/outputs.html
2. http://www.rdg.opengroup.org/public/tech/base/year2000.html
3. http://cr.yp.to/libtai.html
4. http://www.deepsky.com/~merovech/millennium.html
5. http://www.2038bug.com/pivotal_gmtime_r.c.html