YK- 2038 Bug

Vikash Chandra Sharma

Department of Computer Science and Engineering, Kashi Institute of Technology, Varanasi, U.P, India

E-mail: vikash.cse.iu@gmail.com

Abstract

This paper talks about the problem we all will be facing precisely on 19th Jan. 2038. Digital world has been vulnerable by several bugs however, solely a couple of looked as if it would cause a good danger. The bug that got most famed was Y2K. Somehow, we tend to got over it, then, there was Y2K10. It too was resolved and currently we tend to have Y2K38. The Y2K38 bug, if not resolved, will make sure that the predictions that were created for the Y2K bug come true this point. On 19th Jan. 2038 within all UNIX supported Operating system Time and Date would began to work incorrectly. This produces a big problem which is critical for all Software's and basically a threat to embedded systems. Hence afterwards it basically shows possible solutions which we can take to avoid Unix Millennium 2038 problem. By doing research I have developed a patch. Now I have moved this patch to GitHub. This patch can enable the 32bit S/W to use 64bit time_t on a 32bit computer. This patch is still under development but it is current state has been shown in this paper along with all the codes and brief description of each and every function and files. I think to quickies way to solve this problem is by using patch which Software developers who are having 32bit S/w simply needs to include all header files and implementation files within their software and use the function I along with other open source community has made. Coding has been done in C language.

Keywords: *Unix, unix 2038 problem, unix 2038 bug, unix 2038 millennium bug, unix 2038 millennium problem, unix future, time_t, time_t 32bit, embedded computer future, embedded computer 2038, operating system in 2038, operating system in 2038, pathes, unix patch, unix 2038 program*

INTRODUCTION

Sign out from your Yahoo Messenger or Google Talk. Change your Calendar year beyond 2038 say 2040 in your Windows O.S. then log in once again to these messengers what happens? [1, 2].

Messengers would fail! Unix Millennium 2038 Bug is a problem which may seems to be a small but can cause a problem to wide range of devices and Machines. Unix was built in 1969 specifically on 20th April 1969. During that period Unix developer did not know that Software

can live for such a long time (greater than 100 years). But today we have some geeks software available for, e.g., Microsoft office Notepad which simply seems to live even greater than hundreds of year from now on [3–7].

What is Unix Operating system?

Unix officially trademark as UNIX developed in Bell Labs by Geeks-"Ken Thompson, Dennis Ritchie, Brian Kernighan, Douglas McIlroy, Michael Lesk and Joe Ossanna" [8, 9].

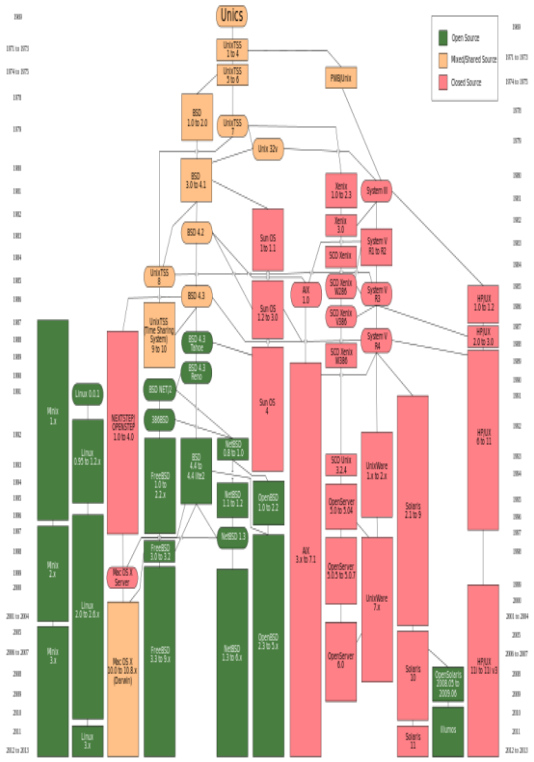

Fig. 1: *Evolution of UNIX.*

So this diagram describes the evolution of Unix. You can clearly see that Mac osx, Linux all have evolved from Unix. However, Windows is not Unix based.

Company/Developer- Ken Thompson, Brian Kernighan, Douglas McIlroy, Dennis Ritchie, Joe Ossanna at Bell Labs.

Programmed In- C and Assembly Language.

OS Family- UNIX.

Source Model- Historically closed source, now some UNIX projects (BSD family and illumos) are open sourced.

Initial Release- April 20, 1969; 44 Years ago.

Available Language(s)- English.

Kernel Type- Monolithic.

Default User Interface- Command Line interface and Graphical (X Window System).

License- Proprietary

Official Website- www.unix.org

What is time_t ?

time_t is a variable of 32 bit. Basically it was developed by UNIX developers during 1970. In all the UNIX based 32bit operating system you will probably see Calendar starting from 1901–2038. This is because time_t is of 32bit which implises it can contain 2^32 int value which is basically 4294967296 days equivalent to 137 years. So time_t can possess years from 1970 till 2107. But time_t is unsigned variable this makes it range +68 and -69. Hence forth its limit is 19th Jan. of 2038 (1970 + 68). 1970–69 gives 1901 hence time_t ranges from 1901–2038(19th Jan.). After 19th Jan. 2038 time_t will get roll back to 1901.

time_t is used by all major software's drivers running on Unix enabled operating system for storing into database for updating there software automatically within certain time period (like Antivirus) and to carry out all other basic features of the software so it's pretty important . After 19th Jan. 2038 it will fail.

What are the major technical areas in which Y2038 issues may exist?

To be specific all those areas which would use 32bit signed time value would fall under this category.

These include:-

- 32bit CPUs
- Firmware/BIOS
- 32bit Operating systems
- Web-based applications and web contents
- Databases
- Computer bus
- Communication protocols
- Communication and networking equipment for e.g., Satellites, Routers, Switches, Mobile electronics, including tablets, netbooks etc.
- Will affect all transportation services.
- Home electronics, such as game consoles.

How big is Y2038? How much will it cost to prepare for it?

To answer this question considers the cost wasted to remove the bog. The digital age before 2000; and now within 2000–2038. In this digital age equipments are a lot

more expensive. Henceforth, experts believe that cost will easily exceed more than 10trillion $. The most vulnerable are embedded computers. This is because embedded computer are time specific. They need to be accurate and embedded computers are used everywhere like we use embedded computers in clock, airplane. If they began to work incorrectly they can pose a huge loss for instance airplane may get crash, clock may stop working correctly. But it is not only limited to embedded computers but also to communication protocols, file system database etc. So Vulnerability of Y2038 problem is really wide and complicated.

What's the worst that could happen?
The worst-case scenario for Y2038 is much more severe than Y2K. Y2K was just a problem related to human interpretation of dates. But Y2038 is a fundamental problem of time related operations for all Unix capable machine. Hence worst-case of Y2038 can be much more severe than Y2K. For instance airplane may get crash, entire global communication can come to halt, Electric power lines can go offline, Cars may stop working or behave erratically and Satellite may stop working and possibly can fall out of orbit. Many embedded devices may malfunction. Fortunately there is plenty of time to resolve these issues.

Can I just recompile my 32-bit program on a 64-bit compiler to fix the problem?
It depends on the targeted machines and nature of programs.
So, probably the problem may not get resolved by simply recompiling the 32-bit program on a 64bit compiler.

How much of my code will likely be affected by Y2038?
Roughly it is estimated that 6% of all source code need to be reviewed. But to say it precisely it depends on the software. Some software which depends on time more frequently would be pretty much more affected than others who are less dependent on time.

Are there any software tools that can help me finding and correcting Y2038 issues?
Yes it is pretty much believed that before 2038 there will be patches available which developers simply need to use inside there software so as to make it work correctly even after 2038. One patch is being made by me and is discussed afterwards in this paper.

How can we prepare and adapt to Y2038 Bug?

First of all there is no universal solution. The cheapest and most effective solution is to use 64bit computers. This will introduce 64bit time_t. 64 time_t can contain a lot more year even greater than 10 times universe life. Hence this will eliminate the problem forever. Looking today's digital media growth it may be possible that during 2038 very few devices will be of 32bit most of them will be of 64bit. But this makes the light fall over to embedded devices. Even today we use 16bit, 32bit embedded devices. Actually embedded devices are really typical to program (Since they 10

For example, the output of this script on Debian GNU/Linux (kernel
2.4.22) (An affected system) are
./2038.pl
Tue January nineteen 03:14:01 2038
Tue January nineteen 03:14:02 2038
Tue January nineteen 03:14:03 2038
Tue January nineteen 03:14:04 2038
Tue January nineteen 03:14:05 2038
Tue January nineteen 03:14:06 2038
Tue January nineteen 03:14:07 2038
Fri Dec thirteen 20:45:52 1901
Fri Dec thirteen 20:45:52 1901
Fri Dec thirteen 20:45:52 1901
Test it currently...

Steps...

1. Login to yahoo traveller.
2. Send instant message to anyone-fine its operating.
3. Now, modification your system dates to 19-Jan-2038, 03:14:07 AM or on top of.
4. Make sure weather your date is modified.
5. Once more send instant message to anyone. Your YM crashes.

MY RESEARCH

Patch

Using internet resources and references I have made a patch. This patch is right now in its development phase because there are new bugs discovered as we test it on different machine.

This is an implementation of POSIX time.h which solves time_t 32bit portability problem. It uses system native code to develop its own time zone and daylight saving time calcutions and thus does not required to be shipped with its own time zone table.

How to use my Patch?

Basically you need to use header files made by me. This header files contains 64bit time representation functions which we can use on 32bit function. So using find and replace dialog box find all the

time_t related function which can pose problems after Y2038 and and replace them with the function made by me.

This patch has two major header files

time64.h	time.h
localtime64_r	localtime_r
localtime64	localtime
gmtime64_r	gmtime_r
gmtime64	gmtime
asctime64_r	asctime_r
asctime64	asctime
ctime64_r	ctime_r
ctime64	ctime
timelocal64	mktime
mktime64	mktime
timegm64	timegm (a GNU extension)

Among these header files major operations performed in header file is in time64.h these operations are Variable used are:-

- days_in_month - No. of days in the month
- julian_days_in_month Sum of days after each consecutive months
- wday_name - weekdays name
- mon_name- Months name
- length_of_year - No. of days in Year
- years_in_gregorian_cycle = 400
- days_in_gregorian_cycle= No. of days in Gregorian cycle
- sec_in_gregorian_cycle= No. of seconds in Gregorian cycle
- MAX_SAFE_YEAR = 2037
- MIN_SAFE_YEAR = 1971
- SOLAR_CYCLE_LENGTH = 28
- safe_years_low[SOLAR_CYCLE_LENGTH]
- CHEAT_DAYS - Total no. of days in 108 years
- CHEAT_YEAR- Total no. of years

Functions used are:-

- IS_LEAP() –
  - is the year leap or not
- USE_SYSTEM_LOCALTIME() –
  - Use system local time
- SHOULD_USE_SYSTEM_GMTIME –
  - To use system gm time if possible
- cmp_date(const struct TIME_PERIOD* left, const struct time_period* right) –
  - Compare two dates. Compares two dates specifically left and right section days like Monday, time (like min.) Ignores things like gmtoffset and dst
- date_in_safe_range(const struct TIME_PERIOD* date, const struct time_period* min) –
  - date range after 2038 which can be used
- const struct time_period* max) -

- Check range of date is it valid or not using cmp_date function
- Time64_T timegm64(const struct TIME_PERIOD *date)-
- returns total sec. according to time64_T
- check_time_period() –
- checks limits of days, min etc.
- Year cycle_offset() –
- The exceptional centuries without leap years cause the cycle to shift by 16. For a given year after y2038 find the the next correct year (Matching Year).

  A matching year...
  - Starts on the same day of the week.
  - Has the same leap year status.

  Over here a previous year must match. For eg. When doing 1st Jan we might end up over to dec. 31st.

  Similarly next year must have the same day of the week.
- safe_year(Year) > returns safe year, i.e., next year which software can use (after 2038)
- copy_time_period_to_TIME_PERIOD 64 (const struct time_period *src, struct TIME_PERIOD *dest) –
- Copies all data of time_period(older time version) like minutes, days, into 64bit time_t
- copy_TIME_PERIOD64_to_time_period (const struct TIME_PERIOD *src, struct time_period *dest) –
- Copies time period 64bit data into time period.
- time_period * fake_localtime_r (const time_t *time, struct time_period *result) –
- Fetches local time from inbuilt time_t and tires to store it within time_t of 64bit
- time_period * fake_gmtime_r (const time_t *time, struct time_period *result)) –
- Fetches local time gm from inbuilt time_t and tires to store it within time_t of 64bit
- Time64_T seconds_between_years (Year leftYear, Year rightYear) -
- returns no. of sec. b/w left and right section of year
- Time64_T mktime64 (struct TIME_PERIOD *input_date) -
- Interprete this input_date to be of time_t 64bit
- Time64_T timelocal64 (struct TIME_PERIOD *date) –
- Interprete date to be of time64_T
- struct TIME_PERIOD *gmtime64_r (const Time64_T *in_time, struct TIME_PERIOD *p) –

- Operating to be performed over here are - If time_t is small then use system gmtime(). returns an date object. This date object contains month, year, day,min...etc. attributes. These attributes are modified according to Gregorian calendar - "On the basis of cycles (cycles is the month/days in Gregorian calendar + 1), whether leap year or not and hence forth some optimization is performed.
- struct TIME_PERIOD *localtime64_r (const Time64_T *time, struct TIME_PERIOD *local_time_period) –
- finds a safe time(Valid time after 2038) within which it would be going to save local_time data char *asctime64_r(const struct TIME_PERIOD* date, char *result) – simply checks when time, week days, months etc. goes out of range like 25th hour and when that happens simply trigger arrayOutOfindexError
- time_t check_date_min
- When time minimum,or when time is underflow
- struct time_period * check_to_time_max
- check exact failure point for maximum value of time instead of exact failure point for maximum value of time
- struct time_period * check_to_time_min --
- check exact failure point for minimum value of time instead of exact failure point for minimum value of time
- void guess_time_limits_from_types(void)
- this function shows that we cant have negative value,,we have to use positive value
- char * time_period_as_json(const struct time_period* date)
- used for time zone
- check_max
- check for maximum
- char *dump = malloc(80 * sizeof(char))
- Creats variable of 80bytes
- double my_difftime
- This function returns d/f of two years
- time_t check_date_max
- Checks When time maximum, or when time is overflow
- time_t check_date_min
- When time minimum,or when time is underflow
- struct time_period * check_to_time_max
- Check exact failure point for maximum value of time instead of exact failure point for maximum value of time

- struct time_period * check_to_time_min
  - Check exact failure point for minimum value of time instead of exact failure point for minimum value of time
- void guess_time_limits_from_types(void)
  - This function shows that we cannot have negative value, we have to use positive value
- char * time_period_as_json(const struct time_period* date)
  - used for time zone

4. Header files

- time64_config.h
  - this header file used for time configuration of the system.
- time.h
  - this file used to declares the time types or structures.
- stdio.h
  - used for standard input/output

math.h

- used for mathematical expression or syntaxes.
- stdlib.h
  - used for genral purpose functions like-macro,dynamic memory management etc.
- string.h
  - that header file used for a character type or for a string.

Source Code

Files

1.time64.c

```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <errno.h>
#include "time64.h"
#include "time64_limits.h"
/* 28 year Julian calendar cycle */
#define SOLAR_CYCLE_LENGTH 28
/* Year cycle from MAX_SAFE_YEAR down. */
static const int safe_yearshigh[SOLAR_CYCLE_LENGTH] = {
    2016, 2017, 2018, 2019,
    2020, 2021, 2022, 2023,
    2024, 2025, 2026, 2027,
    2028, 2029, 2030, 2031,
    2032, 2033, 2034, 2035,
    2036, 2037, 2010, 2011,
    2012, 2013, 2014, 2015
};
/* Year cycle from MIN_SAFE_YEAR up */
static const int safe_yearslow[SOLAR_CYCLE_LENGTH] = {
    1996, 1997, 1998, 1971,
```

```
    1972, 1973, 1974, 1975,
    1976, 1977, 1978, 1979,
    1980, 1981, 1982, 1983,
    1984, 1985, 1986, 1987,
    1988, 1989, 1990, 1991,
    1992, 1993, 1994, 1995,
};
/* This isn't used, but it's handy to look at */
static const int dow_yearstart[SOLAR_CYCLE_LENGTH] = {
    5, 0, 1, 2,    /* 0     2016 - 2019 */
    3, 5, 6, 0,    /* 4  */
    1, 3, 4, 5,    /* 8     1996 - 1998, 1971*/
    6, 1, 2, 3,    /* 12    1972 - 1975 */
    4, 6, 0, 1,    /* 16 */
    2, 4, 5, 6,    /* 20    2036, 2037, 2010, 2011 */
    0, 2, 3, 4     /* 24    2012, 2013, 2014, 2015 */
};
/*Days from epoch on Jan 1st*/
#define CHEAT_DAYS  (1199145600 / 24 / 60 / 60)
#define CHEAT_YEARS 108
#define IS_LEAP(n)      ((!(((n) + 1900) % 400) || (!(((n) + 1900) % 4) && (((n) + 1900) % 100))) != 0)
#define WRAP(a,b,m)     ((a) = ((a) < 0  ) ? ((b)--, (a) + (m)) : (a))
#ifdef USE_SYSTEM_LOCALTIME
#  define SHOULD_USE_SYSTEM_LOCALTIME(a)  (     \
    (a) <= SYSTEM_LOCALTIME_MAX &&  \
    (a) >= SYSTEM_LOCALTIME_MIN     \
)
int cmp_date( const struct TIME_PERIOD* left, const struct time_period* right ) {
    if( left->time_period_year > right->time_period_year )
        return 1;
    else if( left->time_period_year < right->time_period_year )
        return -1;
    If (left->time_period_mon > right->time_period_mon)
        return 1;
    else if (left->time_period_mon < right->time_period_mon)
        return -1;
    If (left->time_period_mday > right->time_period_mday)
        return 1;
    Else if (left->time_period_mday < right->time_period_mday)
        return -1;
```

```
   if ( left->time_period_hour > right->time_period_hour )
      return 1;
   else if( left->time_period_hour < right->time_period_hour )
      return -1;
   if( left->time_period_min > right->time_period_min )
      return 1;
   else if( left->time_period_min < right->time_period_min )
      return -1;
   if( left->time_period_sec > right->time_period_sec )
      return 1;
   else if( left->time_period_sec < right->time_period_sec )
      return -1;
   return 0;
}
/* Checking whether date is inside or not
*/
int date_in_safe_range( const struct TIME_PERIOD* date, const struct time_period* min, const struct time_period* max ) {
   if( cmp_date(date, min) == -1 )
      return 0;
   if( cmp_date(date, max) == 1 )
      return 0;
   return 1;
}
/* timegm() is not in the C or POSIX spec, but it is such a useful
   extension I would be remiss in leaving it out.  Also I need it
   for localtime64 ()
*/
Time64_T timegm64 (const struct TIME_PERIOD *date) {
   Time64_T days    = 0;
   Time64_T seconds = 0;
   Year     year;
   Year        orig_year = (Year) date->time_period_year;
   int      cycles = 0;
   If (orig_year > 100) {
      cycles = (orig_year - 100) / 400;
      orig_year -= cycles * 400;
      days        += (Time64_T)cycles * days_in_gregorian_cycle;
   }
   else if(orig_year < -300 ) {
      Cycles = (orig_year - 100) / 400;
      orig_year -= cycles * 400;
      days        += (Time64_T) cycles * days_in_gregorian_cycle;
   }
   TIME64_TRACE3("# timegm/ cycles: %d, days: %lld, orig_year: %lld\n", cycles, days, orig_year);
   If (orig_year > 70 ) {
      year = 70;
      while( year < orig_year ) {
```

```
        days                    +=
length_of_year[IS_LEAP(year)];
        year++;
      }
   }
   else if ( orig_year < 70 ) {
      year = 69;
      do {
        days                    -=
length_of_year[IS_LEAP(year)];
        year--;
      } while( year >= orig_year );
   }
   days                         +=
julian_days_by_month[IS_LEAP(orig
_year)][date->time_period_mon];
   days += date->time_period_mday -
1;
   seconds = days * 60 * 60 * 24;
   seconds += date->time_period_hour
* 60 * 60;
   seconds += date->time_period_min
* 60;
   seconds += date->time_period_sec;
   return(seconds);
}
static int check_time_period(struct
TIME_PERIOD *time_period)
{
   /* Don't forget leap seconds */
   assert(time_period-
>time_period_sec >= 0);
   assert(time_period-
>time_period_sec <= 61);
   assert(time_period-
>time_period_min >= 0);
   assert(time_period-
>time_period_min <= 59);
   assert(time_period-
>time_period_hour >= 0);
   assert(time_period-
>time_period_hour <= 23);
   assert(time_period-
>time_period_mday >= 1);
   assert(time_period-
>time_period_mday          <=
days_in_month[IS_LEAP(time_period
->time_period_year)][time_period-
>time_period_mon]);
   assert(time_period-
>time_period_mon  >= 0);
   assert(time_period-
>time_period_mon  <= 11);
   assert(time_period-
>time_period_wday >= 0);
   assert(time_period-
>time_period_wday <= 6);
      assert(time_period-
>time_period_yday >= 0);
   assert(time_period-
>time_period_yday          <=
length_of_year[IS_LEAP(time_period-
>time_period_year)]);
```

```
#ifdef
HAS_TIME_PERIOD_TIME_PERIO
D_GMTOFF
   assert(time_period-
>time_period_gmtoff >= -24 * 60 *
60);
   assert(time_period-
>time_period_gmtoff <=  24 * 60 *
60);
#endif
   return 1;
}
static Year cycle_offset(Year year)
{
   const Year start_year = 2000;
   Year year_diff  = year - start_year;
   Year exceptions;
   if( year > start_year )
      year_diff--;
   exceptions  = year_diff / 100;
   exceptions -= year_diff / 400;
   TIME64_TRACE3("#  year:  %lld,
exceptions: %lld, year_diff: %lld\n",
        year, exceptions, year_diff);
   return exceptions * 16;
}
/* For a given year after 2038, pick the
latest possible matching
  year in the 28 year calendar cycle.
  A matching year...
  1) Starts on the same day of the
week.
  2) Has the same leap year status.
  This is so the calendars match up.
  Also the previous year must match.
When doing Jan 1st you might
  wind up on Dec 31st the previous
year when doing a -UTC time zone.
  Finally, the next year must have the
same start day of week.  This
  is for Dec 31st with a +UTC time
zone.
  It doesn't need the same leap year
status since we only care about
  January 1st.
*/
static int safe_year(const Year year)
{
   int safe_year;
   Year year_cycle;
   if( year >= MIN_SAFE_YEAR &&
year <= MAX_SAFE_YEAR ) {
      return (int)year;
   }
   year_cycle  =  year  +
cycle_offset(year);
   if( year < MIN_SAFE_YEAR )
      year_cycle -= 8;
   if( is_exception_century(year) )
      year_cycle += 11;
   /*  Also  xx01  years,  since  the
previous year will be wrong */
   if( is_exception_century(year - 1) )
      year_cycle += 17;
```

```
   year_cycle %= SOLAR_CYCLE_LENGTH;
   if( year_cycle < 0 )
      year_cycle = SOLAR_CYCLE_LENGTH + year_cycle;
   assert( year_cycle >= 0 );
   assert( year_cycle < SOLAR_CYCLE_LENGTH );
   if( year < MIN_SAFE_YEAR )
      safe_year = safe_years_low[year_cycle];
   else if( year > MAX_SAFE_YEAR )
      safe_year = safe_years_high[year_cycle];
   else
      assert(0);
   assert(safe_year <= MAX_SAFE_YEAR && safe_year >= MIN_SAFE_YEAR);

   return safe_year;
}
void copy_time_period_to_TIME_PERIOD64(const struct time_period *src, struct TIME_PERIOD *dest) {
   if( src == NULL ) {
      memset(dest, 0, sizeof(*dest));
   }
   else {
#     ifdef USE_TIME_PERIOD64
         dest->time_period_sec = src->time_period_sec;
         dest->time_period_min = src->time_period_min;
         dest->time_period_hour = src->time_period_hour;
         dest->time_period_mday = src->time_period_mday;
         dest->time_period_mon = src->time_period_mon;
         dest->time_period_year = (Year)src->time_period_year;
         dest->time_period_wday = src->time_period_wday;
         dest->time_period_yday = src->time_period_yday;
         dest->time_period_isdst = src->time_period_isdst;

#        ifdef HAS_TIME_PERIOD_TIME_PERIOD_GMTOFF
            dest->time_period_gmtoff = src->time_period_gmtoff;
#        endif

#        ifdef HAS_TIME_PERIOD_TIME_PERIOD_ZONE
            dest->time_period_zone = src->time_period_zone;
#        endif
```

```
#       else
          /* They're the same type */
          memcpy(dest,                src,
sizeof(*dest));
#       endif
    }
}
void
copy_TIME_PERIOD64_to_time_peri
od(const  struct  TIME_PERIOD  *src,
struct time_period *dest) {
    if( src == NULL ) {
       memset(dest, 0, sizeof(*dest));
    }
    else {
#       ifdef USE_TIME_PERIOD64
          dest->time_period_sec          =
src->time_period_sec;
          dest->time_period_min          =
src->time_period_min;
          dest->time_period_hour         =
src->time_period_hour;
          dest->time_period_mday         =
src->time_period_mday;
          dest->time_period_mon          =
src->time_period_mon;
          dest->time_period_year         =
(int)src->time_period_year;
          dest->time_period_wday         =
src->time_period_wday;
          dest->time_period_yday         =
src->time_period_yday;
          dest->time_period_isdst        =
src->time_period_isdst;

#                                   ifdef
SHAS_TIME_PERIOD_TIME_PERI
OD_GMTOFF
             dest->time_period_gmtoff    =
src->time_period_gmtoff;
#         endif

#                                   ifdef
HAS_TIME_PERIOD_TIME_PERIO
D_ZONE
             dest->time_period_zone      =
src->time_period_zone;
#         endif

#       else
    /* They're the same type */
          memcpy(dest,                src,
    sizeof(*dest));
    #       endif
       }
    }

    struct          time_period          *
    fake_localtime_r(const  time_t  *time,
    struct time_period *result) {
       const          struct          time_period
    *static_result = localtime(time);

       assert(result != NULL);
```

```
    if( static_result == NULL ) {
        memset(result, 0, sizeof(*result));
        return NULL;
    }
    else {
        memcpy(result, static_result, sizeof(*result));
        return result;
    }
}

struct time_period * fake_gmtime_r(const time_t *time, struct time_period *result) {
    const struct time_period *static_result = gmtime(time);

    assert(result != NULL);

    if( static_result == NULL ) {
        memset(result, 0, sizeof(*result));
        return NULL;
    }
    else {
        memcpy(result, static_result, sizeof(*result));
        return result;
    }
}
static Time64_T seconds_between_years(Year left_year, Year right_year) {
    int increment = (left_year > right_year) ? 1 : -1;
    Time64_T seconds = 0;
    int cycles;
    if( left_year > 2400 ) {
        cycles = (left_year - 2400) / 400;
        left_year -= cycles * 400;
        seconds += cycles * seconds_in_gregorian_cycle;
    }
    else if( left_year < 1600 ) {
        cycles = (left_year - 1600) / 400;
        left_year += cycles * 400;
        seconds += cycles * seconds_in_gregorian_cycle;
    }
    while( left_year != right_year ) {
        seconds += length_of_year[IS_LEAP(right_year - 1900)] * 60 * 60 * 24;
        right_year += increment;
    }
    return seconds * increment;
}
Time64_T mktime64(struct TIME_PERIOD *input_date) {
    struct time_period safe_date;
    struct TIME_PERIOD date;
    Time64_T  time;
    Year year = input_date->time_period_year + 1900;
        /* Correct the possibly out of bound input date */
```

```
copy_time_period_to_TIME_PERIOD
64(&safe_date, input_date);
      return time;
   }
   /* Have to make the year safe in
date else it won't fit in safe_date */
   date = *input_date;
   date.time_period_year =
safe_year(year) - 1900;
copy_TIME_PERIOD64_to_time_peri
od(&date, &safe_date);
   time =
(Time64_T)mktime(&safe_date);
   /* Correct the user's possibly out of
bound input date */
copy_time_period_to_TIME_PERIOD
64(&safe_date, input_date);
   time +=
seconds_between_years(year,
(Year)(safe_date.time_period_year +
1900));
   return time;
}
/* Because I think mktime() is a crappy
name */
Time64_T timelocal64(struct
TIME_PERIOD *date) {
   return mktime64(date);
}
struct TIME_PERIOD *gmtime64_r
(const Time64_T *in_time, struct
TIME_PERIOD *p)
{
   int v_time_period_sec,
v_time_period_min,
v_time_period_hour,
v_time_period_mon,
v_time_period_wday;
   Time64_T v_time_period_tday;
   int leap;
   Time64_T m;
   Time64_T time = *in_time;
   Year year = 70;
   int cycles = 0;
   assert(p != NULL);
   /* Use the system gmtime() if time_t
is small enough */
   if(
SHOULD_USE_SYSTEM_GMTIME(
*in_time) ) {
      time_t safe_time =
(time_t)*in_time;
      struct time_period safe_date;
      GMTIME_R(&safe_time,
&safe_date);
copy_time_period_to_TIME_PERIOD
64(&safe_date, p);
      assert(check_time_period(p));
      return p;
   }
#ifdef
HAS_TIME_PERIOD_TIME_PERIO
D_GMTOFF
   p->time_period_gmtoff = 0;
#endif
   p->time_period_isdst  = 0;
```

```
#ifdef HAS_TIME_PERIOD_TIME_PERIOD_ZONE
   p->time_period_zone   = "UTC";
#endif
   v_time_period_sec = (int)(time % 60);
   time /= 60;
   v_time_period_min = (int)(time % 60);
   time /= 60;
   v_time_period_hour = (int)(time % 24);
   time /= 24;
   v_time_period_tday = time;
   WRAP (v_time_period_sec, v_time_period_min, 60);
   WRAP (v_time_period_min, v_time_period_hour, 60);
   WRAP (v_time_period_hour, v_time_period_tday, 24);
   v_time_period_wday = (int)((v_time_period_tday + 4) % 7);
   if (v_time_period_wday < 0)
      v_time_period_wday += 7;
   m = v_time_period_tday;
   if (m >= CHEAT_DAYS) {
      year = CHEAT_YEARS;
      m -= CHEAT_DAYS;
   }
   if (m >= 0) {
      /* Gregorian cycles, this is huge optimization for distant times */
      cycles = (int)(m / (Time64_T) days_in_gregorian_cycle);
      if( cycles ) {
         m -= (cycles * (Time64_T) days_in_gregorian_cycle);
         year += (cycles * years_in_gregorian_cycle);
      }
      /* Years */
      leap = IS_LEAP (year);
      while (m >= (Time64_T) length_of_year[leap]) {
         m -= (Time64_T) length_of_year[leap];
         year++;
         leap = IS_LEAP (year);
      }
      /* Months */
      v_time_period_mon = 0;
      while (m >= (Time64_T) days_in_month[leap][v_time_period_mon]) {
         m -= (Time64_T) days_in_month[leap][v_time_period_mon];
         v_time_period_mon++;
      }
   } else {
      year--;
      /* Gregorian cycles */
      cycles = (int)((m / (Time64_T) days_in_gregorian_cycle) + 1);
      if( cycles ) {
```

```
        m -= (cycles * (Time64_T) days_in_gregorian_cycle);
        year += (cycles * years_in_gregorian_cycle);
      }
      /* Years */
      leap = IS_LEAP (year);
      while (m < (Time64_T) - length_of_year[leap]) {
        m += (Time64_T) length_of_year[leap];
        year--;
        leap = IS_LEAP (year);
      }
      /* Months */
      v_time_period_mon = 11;
      while (m < (Time64_T) - days_in_month[leap][v_time_period_ mon]) {
        m += (Time64_T) days_in_month[leap][v_time_period_ mon];
        v_time_period_mon--;
      }
      m += (Time64_T) days_in_month[leap][v_time_period_ mon];
    }
    p->time_period_year = year;
    if( p->time_period_year != year ) {
#ifdef EOVERFLOW
      errno = EOVERFLOW;
#endif
      return NULL;
    }
    /* Since  m<year */
    p->time_period_mday = (int) m + 1;
    p->time_period_yday = julian_days_by_month[leap][v_time_p eriod_mon] + (int)m;
    p->time_period_sec = v_time_period_sec;
    p->time_period_min = v_time_period_min;
    p->time_period_hour = v_time_period_hour;
    p->time_period_mon = v_time_period_mon;
    p->time_period_wday = v_time_period_wday;
assert(check_time_period(p));
    return p;
}
struct TIME_PERIOD *localtime64_r (const Time64_T *time, struct TIME_PERIOD *local_time_period)
{
    time_t safe_time;
    struct time_period safe_date;
    struct TIME_PERIOD gm_time_period;
    Year orig_year;
    int month_diff;
    assert(local_time_period != NULL);
```

```
  if(
SHOULD_USE_SYSTEM_LOCALTI
ME(*time) ) {
      safe_time = (time_t)*time;
      TIME64_TRACE1("Using
system localtime for %lld\n", *time);
      LOCALTIME_R(&safe_time,
&safe_date);
copy_time_period_to_TIME_PERIOD
64(&safe_date, local_time_period);
assert(check_time_period(local_time_
period));
      return local_time_period;
  }
  if(                gmtime64_r(time,
&gm_time_period) == NULL ) {
      TIME64_TRACE1("gmtime64_r
returned null for %lld\n", *time);
      return NULL;
  }
  orig_year                    =
gm_time_period.time_period_year;
  if
(gm_time_period.time_period_year   >
(2037 - 1900) ||

gm_time_period.time_period_year   <
(1970 - 1900)
    )
  {
      TIME64_TRACE1("Mapping
time_period_year %lld to safe_year",
(Year)gm_time_period.time_period_ye
ar);
gm_time_period.time_period_year   =
safe_year((Year)(gm_time_period.time
_period_year + 1900)) - 1900;
  }
  safe_time                    =
(time_t)timegm64(&gm_time_period);
  if(    LOCALTIME_R(&safe_time,
&safe_date) == NULL ) {
TIME64_TRACE1("localtime_r(%d)
returned NULL\n", (int)safetime);
      return NULL;
  }
copy_time_period_to_TIME_PERIOD
64(&safe_date, local_time_period);

  local_time_period-
>time_period_year = orig_year;
  if(                local_time_period-
>time_period_year != orig_year ) {
TIME64_TRACE2("time_period_year
overflow:   time_period_year   %lld,
orig_year %lld\n",
          (Year)local_time_period-
>time_period_year, (Year)orig_year);

#ifdef EOVERFLOW
      errno = EOVERFLOW;
#endif
      return NULL;
  }
```

```
    month_diff = local_time_period->time_period_mon - gm_time_period.time_period_mon;
    /* When localtime is Dec 31st previous year and
       gmtime is Jan 1st next year.
    */
    if( month_diff == 11 ) {
       local_time_period->time_period_year--;
    }
    if( month_diff == -11 ) {
       local_time_period->time_period_year++;
    }
    if( !ISLEAP(local_time_period->time_period_year) && local_time_period->time_period_yday == 365 )
       local_time_period->time_period_yday--;
assert(check_time_period(local_time_period));
    return local_time_period;
}
int valid_time_period_wday( const struct TIME_PERIOD* date ) {
    if( 0 <= date->time_period_wday && date->time_period_wday <= 6 )
       return 1;
    else
       return 0;
}
int valid_time_period_mon( const struct TIME_PERIOD* date ) {
    if( 0 <= date->time_period_mon && date->time_period_mon <= 11 )
       return 1;
    else
       return 0;
}
char *asctime64_r( const struct TIME_PERIOD* date, char *result ) {
    tim if( !valid_time_period_wday(date) || !valid_time_period_mon(date) )
       return NULL;
    sprintf(result, TIME_PERIOD64_ASCTIME_FORMAT,
       wday_name[date->time_period_wday],
       mon_name[date->time_period_mon],
       date->time_period_mday, date->time_period_hour,
       date->time_period_min, date->time_period_sec,
       1900 + date->time_period_year);

    return result;
}
char *ctime64_r( const Time64_T* time, char* result ) {
    struct TIME_PERIOD date;
    localtime64_r( time, &date );
```

```
    return asctime64_r( &date, result );
}
/* Non-thread safe versions of the above */
struct TIME_PERIOD *localtime64(const Time64_T *time) {
    tzset();
    return localtime64_r(time, &Static_Return_Date);
}
struct TIME_PERIOD *gmtime64(const Time64_T *time) {
    return gmtime64_r(time, &Static_Return_Date);
}
char *asctime64( const struct TIME_PERIOD* date ) {
    return asctime64_r( date, Static_Return_String );
}
char *ctime64( const Time64_T* time ) {
    tzset();
    return asctime64(localtime64(time));
}
```

2. Check_max.c

```
/**
    -> Gonna create a structure date of 80bytes which stores dd mm yy and ss mm hh
    -> time_t is in two forms left AND RIGHT
    ->
**/
#include "time64_config.h"
#include <time.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
struct time_period Test_TIME_PERIOD;
time_t Time_Max;
time_t Time_Min;
time_t Time_Zero = 0;
char *dump_date(const struct time_period *date) {
    char *dump = malloc(80 * sizeof(char));
    sprintf(
        dump,
        "{ %d, %d, %d, %d, %d, %d }",
        date->time_period_sec, date->time_period_min, date->time_period_hour, date->time_period_mday, date->time_period_mon, date->time_period_year
    );
    return dump;
}
/* Visual C++ 2008's difftime() can't do negative times */
double my_difftime(time_t left, time_t right) {
```

```
    double diff = (double)left - (double)right;
    return diff;
}
time_t check_date_max( struct time_period * (*date_func)(const time_t *), const char *func_name ) {
    struct time_period *date;
    time_t time      = Time_Max;
    time_t good_time  = 0;
    time_t time_change = Time_Max;
    /* Binary search for the exact failure point */
    do {
        printf("# Trying %s(%.0f) max...", func_name, my_difftime(time, Time_Zero));
        date = (*date_func)(&time);
        time_change /= 2;
        /* date_func() broke or time_period_year overflowed or time_t overflowed */
        if(date == NULL || date->time_period_year < 69 || time < good_time) {
            printf(" failed\n");
            time -= time_change;
        }
        else {
            printf(" success\n");
            good_time = time;
            time += time_change;
        }
    } while(time_change > 0 && good_time < Time_Max);
    return(good_time);
}
time_t check_date_min( struct time_period * (*date_func)(const time_t *), const char *func_name ) {
    struct time_period *date;
    time_t time      = Time_Min;
    time_t good_time  = 0;
    time_t time_change = Time_Min;
    /* Binary search for the exact failure point */
    do {
        printf("# Trying %s(%.0f) min...", func_name, my_difftime(time, Time_Zero));
        date = (*date_func)(&time);
        time_change /= 2;
        /* date_func() broke or time_period_year overflowed or time_t overflowed */
        if(date == NULL || date->time_period_year > 70 || time > good_time) {
            printf(" failed\n");
            time -= time_change;
        }
        else {
            printf(" success\n");
            good_time = time;
            time += time_change;
        }
```

```
   } while((time_change < 0) && (good_time > Time_Min));
   return(good_time);
}
struct time_period * check_to_time_max( time_t (*to_time)(struct time_period *), const char *func_name,
                struct time_period * (*to_date)(const time_t *) )
{
   time_t round_trip;
   time_t time      = Time_Max;
   time_t good_time  = 0;
   struct time_period *date;
   struct time_period *good_date = malloc(sizeof(struct time_period));
   time_t time_change = Time_Max;
   /* Binary search for the exact failure point */
   do {
      printf("# Trying %s(%.0f) max...", func_name, my_difftime(time, Time_Zero));
      date = (*to_date)(&time);
      round_trip = (*to_time)(date);
      time_change /= 2;
      /* date_func() broke or time_period_year overflowed or time_t overflowed */
      if(time != round_trip) {
         printf(" failed\n");
         time -= time_change;
      }
      else {
         printf(" success\n");
         good_time = time;
         memcpy(good_date, date, sizeof(struct time_period));
         time += time_change;
      }
   } while(time_change > 0 && good_time < Time_Max);
   return(good_date);
}
struct time_period * check_to_time_min( time_t (*to_time)(struct time_period *), const char *func_name,
                struct time_period * (*to_date)(const time_t *) )
{
   time_t round_trip;
   time_t time      = Time_Min;
   time_t good_time  = 0;
   struct time_period *date;
   struct time_period *good_date = malloc(sizeof(struct time_period));
   time_t time_change = Time_Min;
   /* Binary search for the exact failure point */
   do {
      printf("# Trying %s(%.0f) min...", func_name, my_difftime(time, Time_Zero));
      date = (*to_date)(&time);
```

```
        round_trip = (*to_time)(date);
        time_change /= 2;

        /* date_func() broke or
time_period_year overflowed or time_t
overflowed */
        if(time != round_trip) {
            printf(" failed\n");
            time -= time_change;
        }
        else {
            printf(" success\n");
            good_time = time;
            memcpy(good_date,       date,
sizeof(struct time_period));
            time += time_change;
        }
    } while((time_change < 0) &&
(good_time > Time_Min));

    return(good_date);
}
void
guess_time_limits_from_types(void) {
    if( sizeof(time_t) == 4 ) {
        /* y2038 bug, out to 2**31-1 */
        Time_Max =  2147483647;
        Time_Min = -2147483647 - 1;
/*  "C90 doesn't have negative
constants, only
positive ones that have been negated."
*/
    }
    else if( sizeof(time_t) >= 8 ) {
        /* The compiler might warn about
overflowing in the assignments
            below.  Don't worry, these won't
get run in that case */
        if(
sizeof(Test_TIME_PERIOD.time_peri
od_year) == 4 ) {
            /* y2**31-1 bug */
            Time_Max                =
67768036160140799LL;
            Time_Min        =       -
67768036191676800LL;
        }
        else {
            /* All the way out to 2**63-1
*/
            Time_Max                =
9223372036854775807LL;
            Time_Min        =       -
9223372036854775807LL;
        }
    }
    else {
        printf("Weird       sizeof(time_t):
%ld\n", sizeof(time_t));
    }
}
/* Dump a time_period struct as a json
fragment */
char    *   time_period_as_json(const
struct time_period* date) {
```

```
    char *date_json = malloc(sizeof(char) * 512);
#ifdef HAS_TIME_PERIOD_TIME_PERIOD_ZONE
    char zone_json[32];
#endif
#ifdef HAS_TIME_PERIOD_TIME_PERIOD_GMTOFF
    char gmtoff_json[32];
#endif
    sprintf(date_json,
        "\"time_period_sec\": %d, \"time_period_min\": %d, \"time_period_hour\": %d, \"time_period_mday\": %d, \"time_period_mon\": %d, \"time_period_year\": %d, \"time_period_wday\": %d, \"time_period_yday\": %d, \"time_period_isdst\": %d",
        date->time_period_sec, date->time_period_min, date->time_period_hour, date->time_period_mday,
        date->time_period_mon, date->time_period_year, date->time_period_wday, date->time_period_yday, date->time_period_isdst
    );
#ifdef HAS_TIME_PERIOD_TIME_PERIOD_ZONE
    sprintf(zone_json, ", \"time_period_zone\": \"%s\"", date->time_period_zone);
    strcat(date_json, zone_json);
#endif
#ifdef HAS_TIME_PERIOD_TIME_PERIOD_GMTOFF
    sprintf(gmtoff_json, ", \"time_period_gmtoff\": %ld", date->time_period_gmtoff);
    strcat(date_json, gmtoff_json);
#endif
    return date_json;
}
int main(void) {
    time_t gmtime_max;
    time_t gmtime_min;
    time_t localtime_max;
    time_t localtime_min;
#ifdef HAS_TIMEGM
    struct time_period* timegm_max;
    struct time_period* timegm_min;
#endif
    struct time_period* mktime_max;
    struct time_period* mktime_min;
    guess_time_limits_from_types();
    gmtime_max = check_date_max(gmtime, "gmtime");
```

```
    gmtime_min = check_date_min(gmtime, "gmtime");
    localtime_max = check_date_max(localtime, "localtime");
    localtime_min = check_date_min(localtime, "localtime");
#ifdef HAS_TIMEGM
    Time_Max = gmtime_max;
    Time_Min = gmtime_min;
    timegm_max = check_to_time_max(timegm, "timegm", gmtime);
    timegm_min = check_to_time_min(timegm, "timegm", gmtime);
#endif
    Time_Max = localtime_max;
    Time_Min = localtime_min;
    mktime_max = check_to_time_max(mktime, "mktime", localtime);
    mktime_min = check_to_time_min(mktime, "mktime", localtime);
    printf("# system time.h limits, as JSON\n");
    printf("{\n");
    printf("    \"gmtime\": { \"max\": %.0f, \"min\": %0.f },\n",
        my_difftime(gmtime_max, Time Zero),
        my_difftime(gmtime_min, Time Zero)
    );
    printf("    \"localtime\": { \"max\": %.0f, \"min\": %0.f },\n",
        my_difftime(localtime_max, Time Zero),
        my_difftime(localtime_min, Time Zero)
    );
    printf("    \"mktime\": {\n");
    printf("        \"max\": { %s },\n", time_period_as_json(mktime_max));
    printf("        \"min\": { %s }\n", time_period_as_json(mktime_min));
    printf("    }\n");
#ifdef HAS_TIMEGM
    printf("    ,\n");
    printf("    \"timegm\": {\n");
    printf("        \"max\": { %s },\n", time_period_as_json(timegm_max));
    printf("        \"min\": { %s }\n", time_period_as_json(timegm_min));
    printf("    }\n");
#endif
    printf("}\n");
    return 0;
}
```

test_date.c

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

```
int check_date(struct time_period *date, time_t time, char *name) {
   if( date == NULL ) {
      printf("%s(%lld)      return 1;
   }
   else {
      printf("%s(%lld): %s\n", name, (long long)time, asctime(date));
      return 0;
   }
}
int main(int argc, char *argv[]) {
   long long number;
   time_t time;
   struct time_period *localdate;
   struct time_period *gmdate;
   if( argc <= 1 ) {
      printf("usage: %s <time>\n", argv[0]);
      return 1;
   }
   printf("sizeof time_t: %ld, time_period.time_period_year: %ld\n", sizeof(time_t), sizeof(gmdate->time_period_year));
   number = strtoll(argv[1], NULL, 0);
   time = (time_t)number;
   printf("input: %lld, time: %lld\n", number, (long long)time);
   if( time != number ) {
      printf("time_t overflowed\n");
      return 0;
   }
   localdate = localtime(&time);
   gmdate    = gmtime(&time);
   check_date(localdate, time, "localtime");
   check_date(gmdate, time, "gmtime");

   return 0;
}
```

Header Files

1.time.h

```
 #ifndef TIME64_H
#    define TIME64_H
#include <time.h>
#include "time64_config.h"
/* Set our custom types */
typedef INT_64_T      Int64;
typedef Int64         Time64_T;
typedef Int64         Year;
struct TIME_PERIOD64 {
      int    time_period_sec;
      int    time_period_min;
      int    time_period_hour;
      int    time_period_mday;
      int    time_period_mon;
      Year   time_period_year;
      int    time_period_wday;
      int    time_period_yday;
      int    time_period_isdst;
#ifdef HAS_TIME_PERIOD_TIME_PERIOD_GMTOFF
```

```
    long    time_period_gmtoff;
#endif
#ifdef
HAS_TIME_PERIOD_TIME_PERIO
D_ZONE
    char    *time_period_zone;
#endif
};
/* Decide which time_period struct to
use */
#ifdef USE_TIME_PERIOD64
#define                TIME_PERIOD
TIME_PERIOD64
#else
#define                TIME_PERIOD
time_period
#endif
/* Declare public functions */
struct  TIME_PERIOD  *gmtime64_r
(const      Time64_T      *,      struct
TIME_PERIOD *);
struct  TIME_PERIOD  *localtime64_r
(const      Time64_T      *,      struct
TIME_PERIOD *);
struct  TIME_PERIOD  *gmtime64
(const Time64_T *);
struct  TIME_PERIOD  *localtime64
(const Time64_T *);

char *asctime64          (const struct
TIME_PERIOD *);
char *asctime64_r        (const struct
TIME_PERIOD *, char *);
char  *ctime64           (const
Time64_T*);
char  *ctime64_r         (const
Time64_T*, char*);
Time64_T  timegm64  (const struct
TIME_PERIOD *);
Time64_T  mktime64  (struct
TIME_PERIOD *);
Time64_T  timelocal64  (struct
TIME_PERIOD *);
/* Not everyone has gm/localtime_r(),
provide a replacement */
#ifdef HAS_LOCALTIME_R
#    define LOCALTIME_R(clock,
result) localtime_r(clock, result)
#else
#    define LOCALTIME_R(clock,
result) fake_localtime_r(clock, result)
#endif
#ifdef HAS_GMTIME_R
#    define GMTIME_R(clock, result)
gmtime_r(clock, result)
#else
#    define GMTIME_R(clock, result)
fake_gmtime_r(clock, result)
#endif


/* Use a different asctime format
depending on how big the year is */
#ifdef USE_TIME_PERIOD64
  #define
TIME_PERIOD64_ASCTIME_FORM
```

AT "%.3s %.3s%3d %.2d:%.2d:%.2d %lld\n"
#else
#define TIME_PERIOD64_ASCTIME_FORMAT "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n"
#endif
#zndif

2. Time64_config.h

/* Configuration

Define as appropriate for your system.
Sensible defaults provided.
*/
#ifndef TIME64_CONFIG_H
define TIME64_CONFIG_H
/* Debugging
TIME_64_DEBUG
Define if you want debugging messages
*/
/* #define TIME_64_DEBUG */
/* INT_64_T
A 64 bit integer type to use to store time and others.
Must be defined.
*/
#define INT_64_T long long
/* USE_TIME_PERIOD64
Should we use a 64 bit safe replacement for time_period? This will
let you go past year 2 billion but the struct will be incompatible
with time_period. Conversion functions will be provided.
*/
/* #define USE_TIME_PERIOD64 */
/* Availability of system functions.
HAS_GMTIME_R
Define if your system has gmtime_r()
HAS_LOCALTIME_R
Define if your system has localtime_r()
HAS_TIMEGM
Define if your system has timegm(), a GNU extension.
*/
#define HAS_GMTIME_R
#define HAS_LOCALTIME_R
/* #define HAS_TIMEGM */
/* Details of non-standard time_period struct elements.
HAS_TIME_PERIOD_TIME_PERIOD_GMTOFF
True if your time_period struct has a "time_period_gmtoff" element.
A BSD extension.
HAS_TIME_PERIOD_TIME_PERIOD_ZONE

```
  True if your time_period struct has a
"time_period_zone" element.
  A BSD extension.
*/
/*                              #define
HAS_TIME_PERIOD_TIME_PERIO
D_GMTOFF */
/*                              #define
HAS_TIME_PERIOD_TIME_PERIO
D_ZONE */
/* USE_SYSTEM_LOCALTIME
  USE_SYSTEM_GMTIME
  USE_SYSTEM_MKTIME
  USE_SYSTEM_TIMEGM
  Should we use the system functions
if the time is inside their range?
  Your system localtime() is probably
more accurate, but our gmtime() is
  fast and safe.
*/
#define
USE_SYSTEM_LOCALTIME
/*  #define  USE_SYSTEM_GMTIME
*/
#define USE_SYSTEM_MKTIME
/*  #define  USE_SYSTEM_TIMEGM
*/
#endif /* TIME64_CONFIG_H */
3. time64_limits.h
/*
  Maximum and minimum inputs your
system's respective time functions
  can correctly handle.  time64.h will
use your system functions if
  the input falls inside these ranges and
corresponding USE_SYSTEM_*
  constant is defined.
*/
#ifndef TIME64_LIMITS_H
#define TIME64_LIMITS_H
/* Max/min for localtime() */
#define
SYSTEM_LOCALTIME_MAX
2147483647
#define
SYSTEM_LOCALTIME_MIN      -
2147483647-1
/* Max/min for gmtime() */
#define    SYSTEM_GMTIME_MAX
2147483647
#define    SYSTEM_GMTIME_MIN
-2147483647-1
/* Max/min for mktime() */
static    const    struct    time_period
SYSTEM_MKTIME_MAX = {
   7,
   14,
   19,
   18,
   0,
   138,
   1,
   17,
   0
```

```
#ifdef
HAS_TIME_PERIOD_TIME_PERIO
D_GMTOFF
  ,-28800
#endif
#ifdef
HAS_TIME_PERIOD_TIME_PERIO
D_ZONE
  ,"PST"
#endif
};
static const struct time_period
SYSTEM_MKTIME_MIN = {
  52,
  45,
  12,
  13,
  11,
  1,
  5,
  346,
  0
#ifdef
HAS_TIME_PERIOD_TIME_PERIO
D_GMTOFF
  ,-28800
#endif
#ifdef
HAS_TIME_PERIOD_TIME_PERIO
D_ZONE
  ,"PST"
#endif
};
/* Max/min for timegm() */
#ifdef HAS_TIMEGM
static const struct time_period
SYSTEM_TIMEGM_MAX = {
  7,
  14,
  3,
  19,
  0,
```

COCLUSION

Y2038 can really be a serious problem. It can cost more than 10tirllion $ of money. But this problem can be addressed if we began to "try" to remove this bug from right now. Still we have more than 10years to address this problem. Hence at last I would say that if programmers began to address this issue seriously from today there by developing patches or trying to develop some logic through which this situation can be completely eliminated I think this bug would no longer be a serious issue. But for that to take place for eliminating this bug we need to begin right now.

REFERENCES

1. Available at: http://www.unix.org/version2/whatsnew/year2000.htime_per iodl.

2. Available at: http://www.y2038.com/index.htime_periodl.
3. Available at: http://en.wikipedia.org/wiki/Year_2038_problem.
4. Available at: http://computer.howstuffworks.com/question75.htime_period.
5. Available at: http://en.wikipedia.org/wiki/Year_2000_problem.
6. Available at: http://www.howstuffworks.com/y2k.htime_period.
7. A Paper Presentation on 2k38 Problem. By Suprabhat Vamsi Krishna
8. Y2K38 Problem by D.Uday Bhaskar.
9. Y2K38: The BugBy Vishal Singh, Prerna Chaudhary.