Avoid Using POSIX time_t for Telling Time*

John Sauter†

2017-01-27

Abstract

The POSIX data type time_t is defined in a way that leads to errors in application programs when it is used for telling time. Here is how to avoid using it for that purpose.

Keywords: Coordinated Universal Time; UTC; POSIX; time_t.

†System Eyes Computer Store, 20A Northwest Blvd. Ste 345, Nashua, NH 03063-4066, e-mail: John_Sauter@systemeyescomputerstore.com, telephone: (603) 424-1188

1 Definition of time_t

The data type time_t is defined in POSIX[1] as a count of seconds. When telling time, it is the number of seconds since the epoch, which is approximately January 1, 1970. This count of seconds since the epoch is also defined as an encoding of Coordinated Universal time into an integer.

Here is the formal definition:

> 4.16 Seconds Since the Epoch
>
> A value that approximates the number of seconds that have elapsed since the Epoch. A Coordinated Universal Time name (specified in terms of seconds (tm_sec), minutes (tm_min), hours (tm_hour), days since January 1 of the year (tm_yday), and calendar year minus 1900 (tm_year)) is related to a time represented as seconds since the Epoch, according to the expression below.
>
> If the year is < 1970 or the value is negative, the relationship is undefined. If the year is ≥ 1970 and the value is non-negative, the value is related to a Coordinated Universal Time name according to the C-language expression, where tm_sec, tm_min, tm_hour, tm_yday, and tm_year are all integer types:
>
> $$tm\_sec + tm\_min * 60 + tm\_hour * 3600 + tm\_yday * 86400 + (tm\_year - 70) * 31536000 + ((tm\_year - 69)/4) * 86400 - ((tm\_year - 1)/100) * 86400 + ((tm\_year + 299)/400) * 86400$$
>
> The relationship between the actual time of day and the current value for seconds since the Epoch is unspecified.
>
> How any changes to the value of seconds since the Epoch are made to align to a desired relationship with the current actual time is implementation-defined. As represented in seconds since the Epoch, each and every day shall be accounted for by exactly 86 400 seconds.
>
> **Note:** The last three terms of the expression add in a day for each year that follows a leap year starting with the first leap year since the Epoch. The first term adds a day every 4 years starting in 1973, the second subtracts a day back out every 100 years starting in 2001, and the third adds a day back in every 400 years starting in 2001. The divisions in the formula are integer divisions; that is, the remainder is discarded leaving only the integer quotient.

2 Problems

There are problems with this definition.

- It is not defined for years before 1970.

- The relationship between the actual time of day and the current value for seconds since the Epoch is unspecified. This means it can vary from time to time and from one implementation to another, making the relationship meaningless.
- An application using time_t is required to pretend that all days contain exactly 86 400 seconds. This is false: December 31, 2016, for example, contains 86 401 seconds. This pretense leads to application program errors on days that do not contain 86 400 seconds. Because time_t counts seconds almost all the time, application programmers are tempted to use it as though it always counts seconds, leading to errors in the application when it doesn't.
- Because the epoch is undefined, an implementation can change it while an application is running, so that time seems to step backwards. One can forgive an application which fails when this happens.

3 Solutions

The major reason for the problems with time_t is that it is trying to do too much: be a count of seconds and encode Coordinated Universal Time. The first stage in solving the problems is to separate these two jobs.

3.1 Encoding Coordinated Universal Time

A good way to represent Coordinated Universal Time is with the tm data structure. It contains integers for the year, month, day, hour, minute and second. To encode it into an integer you can do this:

$$\begin{aligned} value = {} & (year \times 10000000000) + (month \times 100000000) + \\ & (day \times 1000000) + (hour \times 10000) + (minute \times 100) + second \end{aligned}$$

The value computed above is defined for all years, has a clear relationship to the date and time of day, and does not require an application to pretend that all days have 86 400 seconds. Its values are very different from the value of a time_t, and so are not likely to be mistaken for it, or vice-versa. It does not count seconds, so application programmers will not be tempted to misuse it for that purpose.

3.2 Measuring Time

Application programmers need a way to measure time intevals. If the interval starts and ends during the execution of an application, Monotonic Time can be used, since it is a count of seconds since an arbitrary epoch. Unlike time_t, Monotonic Time does not change its epoch while applications are running, and so Monotonic Time never appears to flow backwards. However, Monotonic Time

is not good for writing to a file or sending over a network to another computer, since each computer will have its own epoch, and that epoch will change when the computer is restarted. For those purposes it is best to use Coordinated Universal Time.

To measure the time between two instants of Coordinated Universal Time requires a table of days whose lengths are not 86 400 seconds. This table can be used to adjust for the number of leap seconds that have occurred during the interval.

4 Implementing the Solutions

The solutions presented above are adequate only if they will allow an application programmer to avoid completely the use of time_t for telling time. To demonstrate that this is possible, I will list all of the POSIX functions that use time_t (or timespec, which contains a time_t) and show how to replace the application functions which use time_t as seconds since the epoch.

4.1 clock_getres

This function returns a timespec, but that timespec describes an interval rather than an instant. The time_t returned in the timespec is a number of seconds rather than seconds since the epoch.

4.2 clock_gettime and gettimeofday

Gettimeofday is equivalent to clock_gettime with clock ID set to CLOCK_REALTIME, except for the time zone, which should not be used.

Clock_gettime with clock ID set to CLOCK_REALTIME can be replaced by a function which returns the current time in a tm structure along with the number of nanoseconds since the last second.

```
/* Subroutine to return the current UTC time in a tm structure
 * and the number of nanoseconds since the start of the last second.
 */
int
time_current_tm_nano (struct tm *current_tm, int *nanoseconds)
{
  struct timex current_timex;
  int adjtimex_result;

  /* Fetch time information from the kernel.  */
  current_timex.status = 0;
  current_timex.modes = 0;
  adjtimex_result = adjtimex (&current_timex);

  /* Format that information into a tm structure.  */
```

```
  gmtime_r (&current_timex.time.tv_sec, current_tm);

  /* If the kernel told us we are in a leap second, increment
   * the seconds value.  This will change it from 59 to 60.  */
  if (adjtimex_result == TIME_OOP)
    {
      current_tm->tm_sec = current_tm->tm_sec + 1;
    }

  /* Return the number of nanoseconds since the start of the last
   * second.  If the kernel's clock is not being controlled by
   * NTP it will return microseconds instead of nanoseconds.  */
  if (current_timex.status & STA_NANO)
    *nanoseconds = current_timex.time.tv_usec;
  else
    *nanoseconds = current_timex.time.tv_usec * 1e3;

  return (0);
}
```

4.3 clock_nanosleep

Clock_nanosleep with clock ID set to CLOCK_REALTIME and with flag TIMER_ABSTIME not set can be replaced by clock_nanosleep with clock ID set to CLOCK_MONOTONIC, since CLOCK_MONOTONIC ticks at same rate as CLOCK_REALTIME.

Clock_nanosleep with clock ID set to CLOCK_REALTIME and with flag TIMER_ABSTIME set can be replaced by a function which accepts a time in a tm structure and a count of nanoseconds since that second.

```
/* Sleep until a specified time.  */
int
time_sleep_until (struct tm *time_tm, int nanoseconds,
                  int variable_length_seconds_before_1972)
{
  long long int seconds_to_sleep;
  struct tm now_tm;
  int now_nanoseconds, ns_to_sleep;
  struct timespec request_timespec;
  struct timespec rem_timespec;
  int return_value;

  /* Fetch the current time.  */
  time_current_tm_nano (&now_tm, &now_nanoseconds);

  /* Compute the number of seconds until the target time.  */
```

```
  seconds_to_sleep =
    time_diff (&now_tm, time_tm,
               variable_length_seconds_before_1972);

  /* Adjust for the number of nanoseconds that have passed since
   * the last second, and the number that should pass after the
   * target second is reached before we awaken.  */
  ns_to_sleep = nanoseconds - now_nanoseconds;
  if (ns_to_sleep < 0)
    {
      ns_to_sleep = ns_to_sleep + 1e9;
      seconds_to_sleep = seconds_to_sleep - 1;
    }

  /* If the target time has already arrived, don't sleep.  */
  if (seconds_to_sleep < 0)
    return (0);
  if ((seconds_to_sleep == 0) && (ns_to_sleep <= 0))
    return (0);

  request_timespec.tv_sec = seconds_to_sleep;
  request_timespec.tv_nsec = ns_to_sleep;

  return_value = nanosleep (&request_timespec, &rem_timespec);
  return (return_value);
}
```

See subsection 4.6 for time_diff and an explanation of variable_length_seconds_before_1972.

4.4 clock_settime

Clock_settime is not used by applications. However, see section 10 for a recommendation that would make setting the time of day during a leap second possible.

4.5 ctime and ctime_r

Ctime and ctime_r can be replaced by a call to localtime followed by a call to asctime. See subsection 4.8 for the substitute for localtime. If asctime isn't flexible enough, use strftime or see subsection 5.4

4.6 difftime

Difftime can be replaced by a function which returns the time in seconds between two times expressed using the tm structure.

```c
/* Subroutine to compute the difference, in seconds, between two
 * instants of Coordinated Universal time.
 *
 * The parameter variable_length_seconds_before_1972 controls
 * the interpretation of seconds before 1972.  Before the invention
 * of the atomic clock in 1955, it was adequate to define the
 * second as 1/86,400 of the length of a mean solar day.
 * By 1960 it had become clear to the scientific world that the
 * Earth's rotation was no longer steady enough for the most
 * accurate clocks, so the definition of a second was changed
 * from Earth-based to atomic-based.
 *
 * Civil time has always been based on the days and the seasons,
 * so it was necessary to make the new second fit into the
 * variable-length day.  By 1972 the current procedure was settled
 * on, in which there are a varying number of fixed-length seconds
 * in each day.
 *
 * Depending on your application's needs, you will want time before
 * 1972 to be measured using modern fixed-length seconds, or
 * the variable-length seconds that were being used at the time.
 * For example, consider this problem: "A beam of light left the
 * Earth on January 1, 1970, at 00:00 UTC.  How far from Earth was
 * it on January 1, 2017, at 00:00 UTC?"  To answer that question
 * you would use modern, fixed-length seconds.  If the problem is
 * "Fredrick was born on February 29, 1856.  How old was he
 * on December 1, 1879?", you would use contemporary seconds.
 *
 * You should use variable-length seconds when you are concerned
 * with the time as told by a clock, such as reckoning birthdays,
 * and fixed-length seconds when you are concerned with activities
 * that are not based on a clock, such as the time it takes an
 * object to fall through a given distance.
 */
long long int
time_diff (struct tm *A_tm, struct tm *B_tm,
           int variable_length_seconds_before_1972)
{
  long long int day_seconds;
  long long int A_exclude, B_include;

  /* Are the two times on the same day?  */
  day_seconds = 0;
  if ((A_tm->tm_year != B_tm->tm_year) ||
      (A_tm->tm_mon != B_tm->tm_mon) ||
```

```
      (A_tm->tm_mday != B_tm->tm_mday))
    {
      /* They are not.  Compute the time in seconds between
       * the beginning of day A and the beginning of day B.  */
      day_seconds =
        diff_day_seconds (A_tm, B_tm,
                          variable_length_seconds_before_1972);
    }

  /* Compute the part of day A that we skip.  */
  A_exclude =
    (A_tm->tm_hour * 3600) + (A_tm->tm_min * 60) + A_tm->tm_sec;

  /* Compute the part of day B that we add.  */
  B_include =
    (B_tm->tm_hour * 3600) + (B_tm->tm_min * 60) + B_tm->tm_sec;

  /* Return the seconds between the days, minus the part of day A
   * that we are not counting, plus the part of day B that we add.
   */
  return (day_seconds - A_exclude + B_include);
}

/* Compute the Julian Day Number corresponding to a specified
 * year, month and day in the Gregorian calender.
 * Here, a Julian Day Number refers to a day as a whole, rather
 * than to a particular moment within a day.  Day 0 is November 24,
 * 4714 BC, and Julian Day Numbers are always integers.  */
int
time_Julian_day_number (int year, int month, int day)
{
  int m, y, prev_days, year_days, leap_days, result;

  /* Adjust the year to start on March 1 to make leap day the last
   * day of the year.  */
  /* m is 0-based, but month is 1-based.  */
  m = month - 3;
  if (m < 0)
    m = m + 12;
  /* We now have m==0 => March, m==1 => April ... m==10 => January,
   * m==11 => February.  */

  /* Count years from -4800.  */
  y = year + 4800;

  /* If the month is January or February, we are in the previous
```

```
   * (March-based) year.  */
  if (month < 3)
    y = y - 1;

  /* Calculate the number of days in the previous months of this
   * year.  */
  prev_days = ((153 * m) + 2) / 5;

  /* Calculate the number of days in previous years excluding
   * leap days.  */
  year_days = 365 * y;

  /* Adjust for an additional day each leap year.
   * We use the Gregorian rule, which means dates before
   * October 15, 1582 are named according to the
   * Proleptic Gregorian calendar.  */
  leap_days = (y / 4) - (y / 100) + (y / 400);

  /* Adjust the base date to be November 24, 4714 BC.  */
  result = year_days + leap_days + prev_days + day - 32045;
  return (result);
}

/* Compute the number of seconds between two days,
 * figuring from the start of day A to the start of day B.  */
static long long int
diff_day_seconds (struct tm *A_tm, struct tm *B_tm,
                  int variable_length_seconds_before_1972)
{
  int A_jdn, B_jdn, A_dtai, B_dtai;
  long long int result;

  /* Compute the Julian Day Number for the two days.  */
  A_jdn = time_Julian_day_number (A_tm->tm_year + 1900,
                                  A_tm->tm_mon + 1,
                                  A_tm->tm_mday);
  B_jdn = time_Julian_day_number (B_tm->tm_year + 1900,
                                  B_tm->tm_mon + 1,
                                  B_tm->tm_mday);

  /* Compute the value of DTAI on each day.  */
  A_dtai = time_DTAI (A_jdn,
                      variable_length_seconds_before_1972);
  B_dtai = time_DTAI (B_jdn,
                      variable_length_seconds_before_1972);
```

```
  /* The result is the number of days between the dates,
   * times 86,400, plus the increase in DTAI.  */
  result =
    ((B_jdn - A_jdn) * (long long int) 86400) +
    (B_dtai - A_dtai);
  return (result);
}

/* List of days which start with a different value of DTAI
 * than the previous day.  Each entry is two integers:
 * the Julian Day Number and the value of DTAI.  */
#include "dtai_table.h"
static int DTAI_table[DTAI_ENTRY_COUNT][2] = {
#include "dtai_table.tab"
};

/* Determine the value of DTAI at the beginning of a day,
 * specified by its Julian Day Number.  */
int
time_DTAI (int day_number,
           int variable_length_seconds_before_1972)
{
  int index, past_limit, future_limit, return_value;
  double days;
  int the_day;

  past_limit = 0;
  future_limit = DTAI_ENTRY_COUNT - 1;

  /* If the application prefers variable-length seconds
   * before 1972, then the value of DTAI for all dates
   * before 1972 is the same as the value for January 1,
   * 1972.  */
  the_day = day_number;
  if ((variable_length_seconds_before_1972 == 1) &&
      (the_day < time_Julian_day_number (1972,1,1)))
    {
      the_day = time_Julian_day_number (1972,1,1);
    }

  if (the_day < DTAI_table[past_limit][0])
    {
      /* The date is before the beginning of the table.
       * Assume 25.2 negative leap seconds per year.  */
      days = DTAI_table[past_limit][0] - the_day;
      return_value =
```

```
      ((int)((double) (days * 25.2) / 365.2425) +
       (double) DTAI_table[past_limit][1]);
    return (return_value);
  }

if (the_day > DTAI_table[future_limit][0])
  {
    /* The date is after the end of the table.
     * Assume 3.34 positive leap seconds per year.  */
    days = the_day - DTAI_table[future_limit][0];
    return_value =
      ((int)((double) (days * 3.34) / 365.2425) +
       (double) DTAI_table[future_limit][1]);
    return (return_value);
  }

/* Do a binary search for the day.  */
for (;;)
  {
    /* Set our index to the midpoint of the current range.  */
    index = (future_limit + past_limit) / 2;
    if (future_limit < past_limit)
      {
        /* The entry is not in the table.
         * Return the entry immediately pastward of
         * the requested day.  */
        return_value = DTAI_table [index][1];
        return (return_value);
      }

    if (the_day < DTAI_table [index][0])
      {
        future_limit = index - 1;
        continue;
      }

    if (the_day > DTAI_table [index][0])
      {
        past_limit = index + 1;
        continue;
      }

    /* We have an exact match.  */
    return_value = DTAI_table [index][1];
    return (return_value);
  }
```

```
  return (0);
}
```

The table of changes in DTAI is too long to present here in full, but here are some of the lines:

```
{2441500, 11}, /* 1 Jul 1972 */
{2441684, 12}, /* 1 Jan 1973 */
{2442049, 13}, /* 1 Jan 1974 */
{2442414, 14}, /* 1 Jan 1975 */
{2442779, 15}, /* 1 Jan 1976 */
{2443145, 16}, /* 1 Jan 1977 */
{2443510, 17}, /* 1 Jan 1978 */
{2443875, 18}, /* 1 Jan 1979 */
{2444240, 19}, /* 1 Jan 1980 */
{2444787, 20}, /* 1 Jul 1981 */
{2445152, 21}, /* 1 Jul 1982 */
{2445517, 22}, /* 1 Jul 1983 */
{2446248, 23}, /* 1 Jul 1985 */
{2447162, 24}, /* 1 Jan 1988 */
{2447893, 25}, /* 1 Jan 1990 */
{2448258, 26}, /* 1 Jan 1991 */
{2448805, 27}, /* 1 Jul 1992 */
{2449170, 28}, /* 1 Jul 1993 */
{2449535, 29}, /* 1 Jul 1994 */
{2450084, 30}, /* 1 Jan 1996 */
{2450631, 31}, /* 1 Jul 1997 */
{2451180, 32}, /* 1 Jan 1999 */
{2453737, 33}, /* 1 Jan 2006 */
{2454833, 34}, /* 1 Jan 2009 */
{2456110, 35}, /* 1 Jul 2012 */
{2457205, 36}, /* 1 Jul 2015 */
{2457755, 37}, /* 1 Jan 2017 */
```

The left number is the Julian Day Number of the day which starts with the new value of DTAI, and the right number is the new value of DTAI.

The full table of changes in DTAI is embedded in this PDF file; see section 7. The table can be generated automatically[5], but the ultimate source of its data is based on manually editing a file when the International Earth Rotation and Reference Systems Service (IERS) issues its Bulletin C[1] announcing the next leap second. See file `extraordinary_days.dat`, also embedded in this PDF file, for details.

[1] `https://www.iers.org/IERS/EN/Publications/Bulletins/bulletins.html`

4.7 gmtime and gmtime_r

Gmtime and gmtime_r convert from time_t to the tm structure, representing Coordinated Universal Time. We don't need this function since our primary representation for time is the tm structure representing Coordinated Universal Time, and we are not using time_t.

4.8 localtime and localtime_r

Localtime and localtime_r convert a time_t into a tm structure that represents local time. We can replace this with a function that takes as input a tm structure that represents Coordinated Universal Time.

```
/* Convert a tm structure containing Coordinated Universal Time
 * to one containing local time.  */
int
time_UTC_to_local (struct tm *coordinated_universal_time,
                   struct tm *local_time_tm,
                   int variable_length_seconds_before_1972)
{
  struct tm utc_time_tm;
  time_t seconds_since_epoch;
  int gmt_offset, hours_offset, minutes_offset, seconds_offset;
  int prev_minute_length;

  /* Compute the number of seconds from the epoch until the
   * beginning of the current minute, since the timegm and
   * localtime functions will not work correctly during a
   * leap second.  */
  time_copy_tm (coordinated_universal_time, &utc_time_tm);
  utc_time_tm.tm_sec = 0;
  seconds_since_epoch = timegm (&utc_time_tm);

  /* Convert to local time in a tm structure.  */
  localtime_r (&seconds_since_epoch, local_time_tm);

  /* Now that we have access to the GMT offset, we do the
   * conversion again, this time taking leap seconds into
   * account.  */
  time_copy_tm (coordinated_universal_time, &utc_time_tm);
  gmt_offset = local_time_tm->tm_gmtoff;

  /* When converting to local time, handle hours, minutes and
   * seconds separately.  That way leap seconds occur at
   * about the same time everywhere, regardless of time zone.
   */
  hours_offset = gmt_offset / 3600;
```

```
minutes_offset = (gmt_offset / 60) - (hours_offset * 60);
seconds_offset =
  gmt_offset - (hours_offset * 3600) - (minutes_offset * 60);

/* Special processing if the time zone is not on a minute
 * boundary.  We delay the leap second until the end of
 * the local minute.  */
if (seconds_offset != 0)
  {
    /* If the time zone is not on a minute boundary,
     * and the minute we are leaving does not have
     * 60 seconds, compensate for it.  */
    prev_minute_length =
      time_length_prev_UTC_minute (&utc_time_tm,
                           variable_length_seconds_before_1972);
    if (prev_minute_length != 60)
      {
        seconds_offset =
          seconds_offset + prev_minute_length - 60;
      }

    /* Always add seconds, rather than subtract.  */
    if (seconds_offset < 0)
      {
        seconds_offset = seconds_offset + 60;
        minutes_offset = minutes_offset - 1;
        if (minutes_offset < 0)
          {
            minutes_offset = minutes_offset + 60;
            hours_offset = hours_offset - 1;
          }
      }
  }

utc_time_tm.tm_hour = utc_time_tm.tm_hour + hours_offset;
utc_time_tm.tm_min = utc_time_tm.tm_min + minutes_offset;
utc_time_tm.tm_sec = utc_time_tm.tm_sec + seconds_offset;

/* Copy the year, month, day, hour, minute and second
 * back to the caller.  */
local_time_tm->tm_year = utc_time_tm.tm_year;
local_time_tm->tm_mon = utc_time_tm.tm_mon;
local_time_tm->tm_mday = utc_time_tm.tm_mday;
local_time_tm->tm_hour = utc_time_tm.tm_hour;
local_time_tm->tm_min = utc_time_tm.tm_min;
local_time_tm->tm_sec = utc_time_tm.tm_sec;
```

```
  /* Make sure all the fields are in their valid ranges.  */
  time_local_normalize (local_time_tm, local_time_tm->tm_sec,
                        variable_length_seconds_before_1972);

  return (0);
}
```

See subsection 5.1 for time_copy_tm, subsection 5.2 for time_length_prev_-UTC_minute, and subsection 5.3 for time_local_normalize.

4.9 mktime

Mktime converts a tm structure that represents local time into a time_t. We can replace this with a function whose output is a tm structure that represents Coordinated Universal Time.

```
/* Convert a tm structure containing local time
 * to one containing Coordinated Universal Time.
 * As a side effect, update the tm_wday, tm_yday,
 * tm_isdst, tm_zone and tm_gmtoff fields in
 * the local_time parameter.  */
int
time_local_to_UTC (struct tm *local_time,
                   struct tm *coordinated_universal_time,
                   int variable_length_seconds_before_1972)
{
  struct tm local_tm;
  struct tm utc_time_tm;
  time_t seconds_since_epoch;
  int gmt_offset;
  int hours_offset, minutes_offset, seconds_offset;

  /* Compute the number of seconds from the epoch to the
   * beginning of the current minute, since the mktime and
   * gmtime functions will not work correctly during a
   * leap second.  */
  time_copy_tm (local_time, &local_tm);
  local_tm.tm_sec = 0;
  seconds_since_epoch = mktime (&local_tm);

  /* Convert to UTC in a tm structure.  */
  gmtime_r (&seconds_since_epoch, &utc_time_tm);

  /* Now that we have access to the GMT offset,
   * do the conversion again, taking leap seconds
   * into account.  */
```

```
gmt_offset = local_tm.tm_gmtoff;
utc_time_tm.tm_year = local_time->tm_year;
utc_time_tm.tm_mon = local_time->tm_mon;
utc_time_tm.tm_mday = local_time->tm_mday;
utc_time_tm.tm_hour = local_time->tm_hour;
utc_time_tm.tm_min = local_time->tm_min;
utc_time_tm.tm_sec = local_time->tm_sec;

/* Add in the hours, minutes and seconds of the offset
 * separately.  That way leap seconds occur at about
 * the same time everywhere.  */
hours_offset = gmt_offset / 3600;
minutes_offset = (gmt_offset / 60) - (hours_offset * 60);
seconds_offset =
  gmt_offset - (minutes_offset * 60) - (hours_offset * 3600);

/* Always add seconds.  */
if (seconds_offset > 0)
  {
    seconds_offset = seconds_offset - 60;
    minutes_offset = minutes_offset + 1;
    if (minutes_offset > 59)
      {
        minutes_offset = minutes_offset - 60;
        hours_offset = hours_offset + 1;
      }
  }

utc_time_tm.tm_hour = utc_time_tm.tm_hour - hours_offset;
utc_time_tm.tm_min = utc_time_tm.tm_min - minutes_offset;
utc_time_tm.tm_sec = utc_time_tm.tm_sec - seconds_offset;

/* Make sure the fields of the tm structure are all in their
 * valid ranges.  */
time_UTC_normalize (&utc_time_tm, utc_time_tm.tm_sec,
                    variable_length_seconds_before_1972);

/* Return the result to the caller.  */
time_copy_tm (&utc_time_tm, coordinated_universal_time);

/* Also return the additional information calculated by mktime.
 */
local_time->tm_wday = local_tm.tm_wday;
local_time->tm_yday = local_tm.tm_yday;
local_time->tm_isdst = local_tm.tm_isdst;
local_time->tm_zone = local_tm.tm_zone;
```

```
  local_time->tm_gmtoff = local_tm.tm_gmtoff;

  return (0);
}
```

See subsection 5.2 for time_UTC_normalize.

4.10 nanosleep

Nanosleep takes a timespec as input, but the time_t portion of that timespec represents the number of seconds to sleep, not an instant of time. Similarly, the timespec output represents the number of seconds remaining in the sleep time. Both time_t values can be considered a count of seconds.

4.11 settimeofday

Settimeofday is not used by applications. However, see section 10 for a recommendation that would make setting the time of day during a leap second possible.

4.12 stat

Stat has three timespec fields: atime, mtime and ctime. The POSIX standard is rather vague on when the fields are updated to the current time, and some file systems have precision less than a second for some of the fields. Your application should not count on these fields being accurate to the second. If you need to compare file times, you should do what rsync does, and implement a modify-window feature, which causes two file times to compare equal if they differ by no more than the specified number of seconds.

Some file systems fill in the nanoseconds part of the timespec field, but you should ignore nanoseconds when comparing because the times may be inaccurate by up to one second, even on modern file systems. However, see section 10 for a recommendation for making the file times accurate.

4.13 time

Instead of returning the current time as a time_t, return it as a tm.

```
/* Subroutine to return the current UTC time in a tm structure.
 * If you need more precision than seconds, see
 * time_current_tm_nano.  */
int
time_current_tm (struct tm *current_tm)
{
  struct timex current_timex;
  int adjtimex_result;
```

```
  /* Fetch time information from the kernel.  */
  current_timex.status = 0;
  current_timex.modes = 0;
  adjtimex_result = adjtimex (&current_timex);

  /* Format that information into a tm structure.  */
  gmtime_r (&current_timex.time.tv_sec, current_tm);

  /* If the kernel told us we are in a leap second, increment
   * the seconds value.  This will change it from 59 to 60.  */
  if (adjtimex_result == TIME_OOP)
    {
      current_tm->tm_sec = current_tm->tm_sec + 1;
    }

  return (0);
}
```

4.14 timegm

Timegm converts a tm structure containing Coordinated Universal Time into a time_t. We don't need it, since we are using the tm structure as our primary representation of time. However, it is sometimes convenient to represent time as a single value instead of a structure.

```
/* Subroutine to convert a time in tm format to an integer.
 * Unlike time_t, this integer has a unique value during a
 * leap second.  */
int
time_tm_to_integer (struct tm *input_tm, long long int *result)
{
  int year, month, day, hour, minute, second;
  long long int result_value;

  year = input_tm->tm_year + 1900;
  month = input_tm->tm_mon + 1;
  day = input_tm->tm_mday;
  hour = input_tm->tm_hour;
  minute = input_tm->tm_min;
  second = input_tm->tm_sec;

  result_value = year;
  result_value = (result_value * 1e2) + month;
  result_value = (result_value * 1e2) + day;
  result_value = (result_value * 1e2) + hour;
  result_value = (result_value * 1e2) + minute;
```

```
  result_value = (result_value * 1e2) + second;
  *result = result_value;

  return (0);
}
```

The following subroutine goes one step further, by including the fraction of a second in the value.

```
/* Subroutine to convert a time in tm format to an integer.
 * Unlike time_t, this integer has a unique value during a
 * leap second.  time_t counts seconds except during a
 * leap second.  This integer never counts seconds, and
 * includes fractions of a second.
 */
int
time_tm_nano_to_integer (struct tm *input_tm, int input_nanoseconds,
                         __int128 *result)
{
  int year, month, day, hour, minute, second;
  __int128 result_value;

  year = input_tm->tm_year + 1900;
  month = input_tm->tm_mon + 1;
  day = input_tm->tm_mday;
  hour = input_tm->tm_hour;
  minute = input_tm->tm_min;
  second = input_tm->tm_sec;

  result_value = year;
  result_value = (result_value * 1e2) + month;
  result_value = (result_value * 1e2) + day;
  result_value = (result_value * 1e2) + hour;
  result_value = (result_value * 1e2) + minute;
  result_value = (result_value * 1e2) + second;
  result_value = (result_value * (__int128) 1e9) +
    (__int128) input_nanoseconds;
  *result = result_value;

  return (0);
}
```

A 128-bit integer cannot be printed with printf, so here is a subroutine to convert it to a string.

```
/* Convert a 128-bit integer to a string.
 * Return value is the number of characters written into the string,
```

```
 * not counting the trailing NUL character.
 * Value is the integer to be converted.
 * Result is a pointer to a string; result_size is its length.
 * The string should be at least 41 characters long, to hold a
 * maximum-sized result.  */

/* This subroutine is dedicated to Steve Russell, who wrote the
 * original recursive decimal print subroutine for the DEC PDP-1
 * in 1964.  */
int
int128_to_string (__int128 value, char *result, int result_size)
{
  __int128 positive_value;
  __int128 remaining_digits;
  int size_remaining;
  int character_count;
  int remaining_count;
  int current_digit;
  char *result_pointer;

  size_remaining = result_size;
  result_pointer = result;
  character_count = 0;

  /* If the value is negative, produce a minus sign
   * and then produce its absolute value.  */
  if (value < 0)
    {
      positive_value = -value;
      if (size_remaining > 0)
        {
          *result_pointer = '-';
          result_pointer = result_pointer + 1;
          size_remaining = size_remaining - 1;
          character_count = character_count + 1;
        }
    }
  else
    {
      positive_value = value;
    }

  /* If the value is zero, produce a 0 and return.  */
  if (positive_value == 0)
    {
      if (size_remaining > 1)
```

```
      {
        *result_pointer = '0';
        result_pointer = result_pointer + 1;
        size_remaining = size_remaining - 1;
        character_count = character_count + 1;
        *result_pointer = 0;
        size_remaining = size_remaining - 1;
      }
    return (character_count);
  }

/* Separate the positive value into its low-order digit and
 * the remaining high-order digits.  We must handle -2**127
 * as a special case because it does not negate.  */

/* If the positive value is less than 0, we have -2**127.
 */
if (positive_value < 0)
  {
    current_digit = 8;
    /* The compiler will not let us say
    remaining_digits =
      (__int128) 170141183460469231731687303715884105728;
      so we build the constant from parts.  */
    remaining_digits = ((__int128) 1701411834604692317 *
                        (__int128) 1e19)
      + (__int128) 3168730371588410572;
  }
else
  {
    current_digit = positive_value % 10;
    remaining_digits = positive_value / 10;
  }

/* If we have any high-order digits, produce them.  We can call
 * this subroutine recursively because we know the value we are
 * passing is positive and is less than the positive value passed
 * to us, so the recursion will terminate.  */
if (remaining_digits != 0)
  {
    remaining_count =
      int128_to_string (remaining_digits,
                        result_pointer, size_remaining);
    result_pointer = result_pointer + remaining_count;
    size_remaining = size_remaining - remaining_count;
    character_count = character_count + remaining_count;
```

```
    }

  /* Having produced the digits of higher order than the low-order
   * digit of this value, append that low-order digit to the string.
   */
  if (size_remaining > 1)
    {
      *result_pointer = '0' + current_digit;
      result_pointer = result_pointer + 1;
      size_remaining = size_remaining - 1;
      character_count = character_count + 1;

      /* If we are being called recursively, the following NUL byte
       * will be overwritten by the next digit.  */
      *result_pointer = 0;
      size_remaining = size_remaining - 1;
    }

  /* By returning the number of characters we wrote into the string,
   * not counting the trailing NUL byte, we make the recursive call
   * simpler since it has to find the end of the string.  */
  return (character_count);
}
```

5 Beyond POSIX

Just being able to do all the POSIX functions isn't enough to persuade application programmers to abandon time_t for telling time. Application programmers need a net benefit to make changing worthwhile. To sweeten the solution, here are some subroutines to manipulate a time value.

5.1 Copy a time

Here is how to copy a time value from one tm structure to another.

```
/* Copy a time from one tm structure to another.  */
int
time_copy_tm (struct tm *in_tm, struct tm *out_tm)
{
  memset (out_tm, 0, sizeof (*out_tm));
  out_tm->tm_year = in_tm->tm_year;
  out_tm->tm_mon = in_tm->tm_mon;
  out_tm->tm_mday = in_tm->tm_mday;
  out_tm->tm_hour = in_tm->tm_hour;
  out_tm->tm_min = in_tm->tm_min;
  out_tm->tm_sec = in_tm->tm_sec;
```

```
  out_tm->tm_yday = in_tm->tm_yday;
  out_tm->tm_wday = in_tm->tm_wday;
  out_tm->tm_isdst = in_tm->tm_isdst;
  out_tm->tm_gmtoff = in_tm->tm_gmtoff;
  out_tm->tm_zone = in_tm->tm_zone;

  return (0);
}
```

5.2 Adding years, months, days, hours, minutes or seconds

If you have a time value stored in a tm structure, and it represents Coordinated Universal Time, you can add to the year, month, day, hour, minute or second field. You can decrease these fields by adding a negative value. When the target date is invalid, such as February 31, you can choose whether to round towards the future or towards the past to get a valid date.

```
/* Add to a field of a tm structure, and then make all the fields
 * have valid values.  */

/* Rounding Mode is used when a modification to a field has caused
 * another field to be invalid.  For example, if you start with
 * October 31 and increase the month by one, you have November 31,
 * which does not exist.  If Rounding Mode is -1, the result is
 * November 30.  If Rounding Mode is +1, the result is December 1.
 * Another example is birthdays on February 29.  If you were born
 * on February 29, 1996, your twentieth birthday is on February 29,
 * 2016.  What day is your twenty-first birthday?  Start with
 * February 29, 1996 and increase the year by twenty-one
 * to get February 29, 2017, then set Rounding Mode to +1
 * if you want March 1, 2017, or -1 if you want February 28, 2017.
 *
 * Like months, not all minutes are the same length.  If you start
 * with December 31, 2016, 23:59:60, and decrease the minutes by one
 * you get December 31, 2016, 23:58:60, which does not exist.
 * If Rounding Mode is +1 you get December 31, 2016, 23:59:00.  If
 * Rounding Mode is -1 you get December 31, 2016, 23:58:59.
 */

/* After changing the day, make sure the seconds are correct.  */
static int
time_UTC_adjust_seconds (struct tm *time_tm, int rounding_mode,
                         int variable_length_seconds_before_1972)
{
  /* The seconds can be invalid if we are at the end of the day and
```

```
  * the day we came from was longer than this day.  This can happen
  * if the day we came from was 86,401 seconds long, or if the
  * day we came to was 86,399 seconds long, or both.  */
 if (time_tm->tm_sec >=
     time_length_UTC_minute (time_tm,
                             variable_length_seconds_before_1972))
   {
     switch (rounding_mode)
       {
       case 1:
         /* Round up to the first second of the next day.
          * This will often be the first second of the next year.
          */
         time_tm->tm_sec = 0;
         time_tm->tm_min = time_tm->tm_min + 1;
         if (time_tm->tm_min > 59)
           {
             time_tm->tm_min = 0;
             time_tm->tm_hour = time_tm->tm_hour + 1;
             if (time_tm->tm_hour > 23)
               {
                 time_tm->tm_hour = 0;
                 time_tm->tm_mday = time_tm->tm_mday + 1;
                 if (time_tm->tm_mday > time_length_month (time_tm))
                   {
                     time_tm->tm_mday = 1;
                     time_tm->tm_mon = time_tm->tm_mon + 1;
                     if (time_tm->tm_mon > 11)
                       {
                         time_tm->tm_mon = 0;
                         time_tm->tm_year = time_tm->tm_year + 1;
                       }
                   }
               }
           }
         break;

       case -1:
         /* Round doum to the last second of this day.  */
         time_tm->tm_sec =
           time_length_UTC_minute (time_tm,
                                   variable_length_seconds_before_1972)
           - 1;
         break;

       default:
```

```
          break;
        }
    }
  return (0);
}

/* Add a specified number of years to a specified time.  */
int
time_UTC_add_years (struct tm *time_tm, int addend,
                    int rounding_mode,
                    int variable_length_seconds_before_1972)
{
  int return_value;

  time_tm->tm_year = time_tm->tm_year + addend;

  /* We have just changed the year.  If the day of the month is
   * outside its valid range, which can only be the case for
   * February 29, round it up to March 1 or down to February 28,
   * depending on the rounding mode.
   */
  if (time_tm->tm_mday > time_length_month (time_tm))
    {
      switch (rounding_mode)
        {
        case 1:
          /* Round up to March 1.  */
          time_tm->tm_mday = 1;
          time_tm->tm_mon = time_tm->tm_mon + 1;
          while (time_tm->tm_mon > 11)
            {
              time_tm->tm_mon = time_tm->tm_mon - 12;
              time_tm->tm_year = time_tm->tm_year + 1;
            }
          break;

        case -1:
          /* Round down to February 28.  */
          time_tm->tm_mday = time_length_month (time_tm);
          break;

        default:
          break;
        }
    }
```

```
  /* Adjust the value of the seconds field to be valid.  */
  return_value =
    time_UTC_adjust_seconds (time_tm, rounding_mode,
                             variable_length_seconds_before_1972);
  if (return_value != 0)
    return (return_value);

  return_value =
    time_UTC_normalize (time_tm, time_tm->tm_sec,
                        variable_length_seconds_before_1972);
  return (return_value);
}

/* Add a specified number of months to the time.
 * To subtract months, make the amount to add negative.  */
int
time_UTC_add_months (struct tm *time_tm, int addend,
                     int rounding_mode,
                     int variable_length_seconds_before_1972)
{
  int return_value;

  time_tm->tm_mon = time_tm->tm_mon + addend;

  /* Get the months value into its valid range, compensating by
   * adjusting the years.  */
  while (time_tm->tm_mon < 0)
    {
      time_tm->tm_mon = time_tm->tm_mon + 12;
      time_tm->tm_year = time_tm->tm_year - 1;
    }
  while (time_tm->tm_mon > 11)
    {
      time_tm->tm_mon = time_tm->tm_mon - 12;
      time_tm->tm_year = time_tm->tm_year + 1;
    }

  /* If we are at the end of the month, we may have to adjust
   * the day to the last day of this month or the first day
   * of the next.  */
  if (time_tm->tm_mday > time_length_month (time_tm))
    {
      switch (rounding_mode)
        {
        case 1:
          time_tm->tm_mday = 1;
```

```
          time_tm->tm_mon = time_tm->tm_mon + 1;
          if (time_tm->tm_mon > 11)
            {
              /* This is only a formality--December has 31 days
               * so we will never have to round up to January of
               * the next year.  */
              time_tm->tm_mon = time_tm->tm_mon - 12;
              time_tm->tm_year = time_tm->tm_year + 1;
            }
          break;

        case -1:
          time_tm->tm_mday = time_length_month (time_tm);
          break;

        default:
          break;
        }
    }

  /* Make sure the value of the seconds field is valid.  */
  return_value =
    time_UTC_adjust_seconds (time_tm, rounding_mode,
                             variable_length_seconds_before_1972);
  if (return_value != 0)
    return (return_value);

  return_value =
    time_UTC_normalize (time_tm, time_tm->tm_sec,
                        variable_length_seconds_before_1972);
  return (return_value);
}

/* Add a specified number of days to a specified time.  */
int
time_UTC_add_days (struct tm *time_tm, int addend,
                   int rounding_mode,
                   int variable_length_seconds_before_1972)
{
  int return_value;

  time_tm->tm_mday = time_tm->tm_mday + addend;

  /* Go through the months one at a time because they have
   * non-uniform lengths.  */
  while (time_tm->tm_mday < 1)
```

```
    {
      /* We are backing into the previous month.  */
      time_tm->tm_mon = time_tm->tm_mon - 1;
      if (time_tm->tm_mon < 0)
        {
          time_tm->tm_mon = time_tm->tm_mon + 12;
          time_tm->tm_year = time_tm->tm_year - 1;
        }
      time_tm->tm_mday =
        time_tm->tm_mday + time_length_month (time_tm);
    }

  while (time_tm->tm_mday > time_length_month (time_tm))
    {
      /* We are moving forward into future months.  */
      time_tm->tm_mday =
        time_tm->tm_mday - time_length_month (time_tm);
      time_tm->tm_mon = time_tm->tm_mon + 1;
      if (time_tm->tm_mon > 11)
        {
          time_tm->tm_mon = time_tm->tm_mon - 12;
          time_tm->tm_year = time_tm->tm_year + 1;
        }
    }

  /* Make sure the value of the seconds field is valid.  */
  return_value =
    time_UTC_adjust_seconds (time_tm, rounding_mode,
                             variable_length_seconds_before_1972);
  if (return_value != 0)
    return (return_value);

  return_value =
    time_UTC_normalize (time_tm, time_tm->tm_sec,
                        variable_length_seconds_before_1972);
  return (return_value);
}

/* Add a specified number of hours to the specified time.  */
int
time_UTC_add_hours (struct tm *time_tm, int addend,
                    int rounding_mode,
                    int variable_length_seconds_before_1972)
{
  int return_value;
```

```
time_tm->tm_hour = time_tm->tm_hour + addend;

/* If the hours field is out of range, adjust it.  */
while (time_tm->tm_hour < 0)
  {
    time_tm->tm_hour = time_tm->tm_hour + 24;
    time_tm->tm_mday = time_tm->tm_mday - 1;
    if (time_tm->tm_mday < 1)
      {
        time_tm->tm_mday =
          time_tm->tm_mday + time_length_prev_month (time_tm);
        time_tm->tm_mon = time_tm->tm_mon - 1;
        if (time_tm->tm_mon < 0)
          {
            time_tm->tm_mon = time_tm->tm_mon + 12;
            time_tm->tm_year = time_tm->tm_year - 1;
          }
      }
  }

while (time_tm->tm_hour > 23)
  {
    time_tm->tm_hour = time_tm->tm_hour - 24;
    time_tm->tm_mday = time_tm->tm_mday + 1;
    if (time_tm->tm_mday > time_length_month (time_tm))
      {
        time_tm->tm_mday =
          time_tm->tm_mday - time_length_month (time_tm);
        time_tm->tm_mon = time_tm->tm_mon + 1;
        if (time_tm->tm_mon > 11)
          {
            time_tm->tm_mon = time_tm->tm_mon - 12;
            time_tm->tm_year = time_tm->tm_year + 1;
          }
      }
  }

/* Make sure the seconds field is valid.  */
return_value =
  time_UTC_adjust_seconds (time_tm, rounding_mode,
                           variable_length_seconds_before_1972);
if (return_value != 0)
  return (return_value);

return_value =
  time_UTC_normalize (time_tm, time_tm->tm_sec,
```

```
                        variable_length_seconds_before_1972);
  return (return_value);
}

/* Add a specified number of minutes to the specified time.  */
int
time_UTC_add_minutes (struct tm *time_tm, int addend,
                      int rounding_mode,
                      int variable_length_seconds_before_1972)
{
  int return_value;

  time_tm->tm_min = time_tm->tm_min + addend;

  /* If the minutes field is out of range, adjust it.  */
  while (time_tm->tm_min < 0)
    {
      time_tm->tm_min = time_tm->tm_min + 60;
      time_tm->tm_hour = time_tm->tm_hour - 1;
      if (time_tm->tm_hour < 0)
        {
          time_tm->tm_hour = time_tm->tm_hour + 24;
          time_tm->tm_mday = time_tm->tm_mday - 1;
          if (time_tm->tm_mday < 1)
            {
              time_tm->tm_mday =
                time_tm->tm_mday + time_length_prev_month (time_tm);
              time_tm->tm_mon = time_tm->tm_mon - 1;
              if (time_tm->tm_mon < 0)
                {
                  time_tm->tm_mon = time_tm->tm_mon + 12;
                  time_tm->tm_year = time_tm->tm_year - 1;
                }
            }
        }
    }

  while (time_tm->tm_min > 59)
    {
      time_tm->tm_min = time_tm->tm_min - 60;
      time_tm->tm_hour = time_tm->tm_hour + 1;
      if (time_tm->tm_hour > 23)
        {
          time_tm->tm_hour = time_tm->tm_hour - 24;
          time_tm->tm_mday = time_tm->tm_mday + 1;
          if (time_tm->tm_mday > time_length_month (time_tm))
```

```
            {
              time_tm->tm_mday =
                time_tm->tm_mday - time_length_month (time_tm);
              time_tm->tm_mon = time_tm->tm_mon + 1;
              if (time_tm->tm_mon > 11)
                {
                  time_tm->tm_mon = time_tm->tm_mon - 12;
                  time_tm->tm_year = time_tm->tm_year + 1;
                }
            }
        }
    }

  /* Make sure the seconds field is valid.  */
  return_value =
    time_UTC_adjust_seconds (time_tm, rounding_mode,
                             variable_length_seconds_before_1972);
  if (return_value != 0)
    return (return_value);

  return_value =
    time_UTC_normalize (time_tm, time_tm->tm_sec,
                        variable_length_seconds_before_1972);
  return (return_value);
}

/* Add a specified number of seconds to a time.  */
int
time_UTC_add_seconds (struct tm *time_tm,
                      long long int add_seconds,
                      int variable_length_seconds_before_1972)
{
  long long int add_nanoseconds;
  long long int nanoseconds;
  int return_value;

  /* Use time_UTC_add_seconds_ns with nanoseconds == 0.  */
  nanoseconds = 0;
  add_nanoseconds = 0;
  return_value =
    time_UTC_add_seconds_ns (time_tm, &nanoseconds,
                             add_seconds, add_nanoseconds,
                             variable_length_seconds_before_1972);
  return (return_value);
}
```

```
/* Add a specified number of seconds and nanoseconds to a time.
 */
int
time_UTC_add_seconds_ns (struct tm *time_tm,
                         long long int *nanoseconds,
                         long long int add_seconds,
                         long long int add_nanoseconds,
                         int variable_length_seconds_before_1972)
{
  long long int accumulated_seconds;
  long long int accumulated_nanoseconds;
  int return_value;

  /* Bring the nanoseconds into its proper range, compensating
   * by adjusting seconds.  */
  accumulated_seconds = time_tm->tm_sec + add_seconds;
  accumulated_nanoseconds = *nanoseconds + add_nanoseconds;
  while (accumulated_nanoseconds >= 1e9)
    {
      accumulated_seconds = accumulated_seconds + 1;
      accumulated_nanoseconds = accumulated_nanoseconds - 1e9;
    }
  while (accumulated_nanoseconds < 0)
    {
      accumulated_seconds = accumulated_seconds - 1;
      accumulated_nanoseconds = accumulated_nanoseconds + 1e9;
    }

  /* The caller can increase or decrease the seconds by a large
   * amount to navigate the calendar by seconds.  Adjust the year,
   * month, day of the month, hour and minute fields to get the
   * seconds field into its valid range, and thereby name the
   * designated second using the Gregorian calendar.
   * Note that time_UTC_normalize takes the seconds value in a
   * long long int parameter and ignores the tm_sec field of time_tm.
   * This is to allow the number of seconds to be very large or
   * very negative.  When time_UTC_normalize is complete
   * it stores the adjusted value of the seconds parameter into
   * time_tm->tm_sec.  */
  return_value =
    time_UTC_normalize (time_tm, accumulated_seconds,
                        variable_length_seconds_before_1972);
  *nanoseconds = accumulated_nanoseconds;
  return (return_value);
}
```

The routines above use various time_length subroutines to determine the length of the current or previous month or minute.

```
/* Return the length of a month.  Most months have fixed lengths,
 * but February has either 28 or 29 days, depending on the year.
 */
int
time_length_month (struct tm *time_tm)
{
  int year;

  switch (time_tm->tm_mon)
    {
    case 0: /* January */
      return (31);
      break;

    case 1: /* February */
      year = time_tm->tm_year + 1900;
      if ((year % 4) != 0)
        return (28);

      /* The year is a multiple of 4.  */
      if ((year % 100) != 0)
        {
          /* The year is not a century year, so it is a leap year.
           */
          return (29);
        }

      /* The year is a century year.  */
      if ((year % 400) == 0)
        {
          /* The four-century years are leap years.  */
          return (29);
        }
      else
        {
          /* The century years that are not four-century years
           * are not leap years.  */
          return (28);
        }
      break;

    case 2: /* March */
      return (31);
```

```
      break;

    case 3: /* April */
      return (30);
      break;

    case 4: /* May */
      return (31);
      break;

    case 5: /* June */
      return (30);
      break;

    case 6: /* July */
      return (31);
      break;

    case 7: /* August */
      return (31);
      break;

    case 8: /* September */
      return (30);
      break;

    case 9: /* October */
      return (31);
      break;

    case 10: /* November */
      return (30);
      break;

    case 11: /* December */
      return (31);
      break;

    default:
      return (0);
      break;
    }

  return (0);
}
```

```c
/* Return the length of the previous month, which might be
 * the last month of the previous year.  */
int
time_length_prev_month (struct tm *time_tm)
{

  struct tm prev_month_tm;
  int return_value;

  /* Compute the previous month.  */
  time_copy_tm (time_tm, &prev_month_tm);
  prev_month_tm.tm_mon = time_tm->tm_mon - 1;
  if (prev_month_tm.tm_mon < 0)
    {
      prev_month_tm.tm_mon = prev_month_tm.tm_mon + 12;
      prev_month_tm.tm_year = prev_month_tm.tm_year - 1;
    }
  return_value = time_length_month (&prev_month_tm);
  return (return_value);
}

/* Return the length of this UTC minute.  */
int
time_length_UTC_minute (struct tm *time_tm,
                        int variable_length_seconds_before_1972)
{

  int JDN_today, JDN_tomorrow, DTAI_today, DTAI_tomorrow;

  /* All minutes are 60 seconds in length except possibly
   * the last minute of the UTC day.  */
  if ((time_tm->tm_hour != 23) || (time_tm->tm_min != 59))
    return (60);

  /* Calculate the Julian Day Number of the specified date.  */
  JDN_today = time_Julian_day_number (time_tm->tm_year + 1900,
                                      time_tm->tm_mon + 1,
                                      time_tm->tm_mday);

  /* Calculate the number of seconds between UTC and TAI,
   * known as DTAI, for this date.  */
  DTAI_today = time_DTAI (JDN_today,
                          variable_length_seconds_before_1972);

  /* Do the same for tomorrow.  */
  JDN_tomorrow = JDN_today + 1;
```

```
  DTAI_tomorrow = time_DTAI (JDN_tomorrow,
                             variable_length_seconds_before_1972);

  /* If the value of DTAI did not change between the beginning
   * of the day today and the beginning of the day tomorrow,
   * then today has 86,400 seconds and the last minute of the day
   * has 60 seconds.
   */
  if (DTAI_today == DTAI_tomorrow)
    return (60);

  /* Otherwise, modify the length of last minute of the day to
   * account for the increase or decrease in DTAI.  */
  return (60 + DTAI_tomorrow - DTAI_today);
}

/* Return the number of seconds in the previous UTC minute.  */
int
time_length_prev_UTC_minute (struct tm *time_tm,
                             int variable_length_seconds_before_1972)
{
  struct tm prev_minute_tm;
  int return_value;

  /* Compute the previous minute.  If this minute is the first
   * minute of he year, the previous minute will be the last
   * minute of the previous year.  */
  time_copy_tm (time_tm, &prev_minute_tm);
  prev_minute_tm.tm_min = time_tm->tm_min - 1;
  if (prev_minute_tm.tm_min < 0)
    {
      prev_minute_tm.tm_min = prev_minute_tm.tm_min + 60;
      prev_minute_tm.tm_hour = prev_minute_tm.tm_hour - 1;
      if (prev_minute_tm.tm_hour < 0)
        {
          prev_minute_tm.tm_hour = prev_minute_tm.tm_hour + 24;
          prev_minute_tm.tm_mday = prev_minute_tm.tm_mday - 1;
          if (prev_minute_tm.tm_mday < 1)
            {
              prev_minute_tm.tm_mday =
                time_length_prev_month (time_tm);
              prev_minute_tm.tm_mon = prev_minute_tm.tm_mon - 1;
              if (prev_minute_tm.tm_mon < 0)
                {
                  prev_minute_tm.tm_mon =
                    prev_minute_tm.tm_mon + 12;
```

```
                prev_minute_tm.tm_year =
                  prev_minute_tm.tm_year - 1;
              }
            }
          }
        }

  return_value =
    time_length_UTC_minute (&prev_minute_tm,
                            variable_length_seconds_before_1972);
  return (return_value);
}

/* Return the length of this local minute.  */
int
time_length_local_minute (struct tm *time_tm,
                          int variable_length_seconds_before_1972)
{
  /* Modern time zones are mostly 60 minutes offset from UTC.
   * Those that are not are mostly 30 minutes offset.  All are
   * offset by an integral number of minutes.  This makes it
   * easy to determine which local minute contains a
   * leap second: it is the one that starts at the same time
   * as the UTC minute that contains a leap second.
   * However, in the past there were some time zones that
   * were not offset from UTC by an integral number of minutes.
   * Since we are extending leap seconds into the past, we are
   * forced to decide which local minute, not starting at the
   * same time as any UTC minute, should contain the leap second.
   * I have chosen the minute containing the leap second to be
   * the local minute that starts during the UTC minute
   * containing the leap second.  However, I defer the leap
   * to the end of the local minute, since it is clear that
   * second number 60 is an additional second, but it is not
   * clear what to call a second that is added earlier in the
   * minute.
   *
   * This problem is somewhat artificial, since it is only
   * because we are pretending that leap seconds existed before
   * time zones that the issue even arises.
   */
  struct tm utc_time_tm;
  struct tm local_tm;
  int return_val;

  /* Convert the local time to UTC and return the length of
```

```
   * the UTC minute corresponding to the local
   * time minute.  */
  time_copy_tm (time_tm, &local_tm);
  local_tm.tm_sec = 0;
  time_local_to_UTC (&local_tm, &utc_time_tm,
                     variable_length_seconds_before_1972);
  return_val =
    time_length_UTC_minute (&utc_time_tm,
                            variable_length_seconds_before_1972);
  return (return_val);
}

/* Return the number of seconds in the previous local minute.  */
int
time_length_prev_local_minute (struct tm *time_tm,
                               int variable_length_seconds_before_1972)
{
  struct tm prev_minute_tm;
  int return_value;

  /* Compute the previous minute.  If this minute is the first
   * minute of the year, the previous minute will be the last
   * minute of the previous year.  */
  time_copy_tm (time_tm, &prev_minute_tm);
  prev_minute_tm.tm_min = time_tm->tm_min - 1;
  if (prev_minute_tm.tm_min < 0)
    {
      prev_minute_tm.tm_min = prev_minute_tm.tm_min + 60;
      prev_minute_tm.tm_hour = prev_minute_tm.tm_hour - 1;
      if (prev_minute_tm.tm_hour < 0)
        {
          prev_minute_tm.tm_hour = prev_minute_tm.tm_hour + 24;
          prev_minute_tm.tm_mday = prev_minute_tm.tm_mday - 1;
          if (prev_minute_tm.tm_mday < 1)
            {
              prev_minute_tm.tm_mday =
                time_length_prev_month (time_tm);
              prev_minute_tm.tm_mon = prev_minute_tm.tm_mon - 1;
              if (prev_minute_tm.tm_mon < 0)
                {
                  prev_minute_tm.tm_mon =
                    prev_minute_tm.tm_mon + 12;
                  prev_minute_tm.tm_year =
                    prev_minute_tm.tm_year - 1;
                }
            }
```

```
        }
    }

  return_value =
    time_length_local_minute (&prev_minute_tm,
                              variable_length_seconds_before_1972);
  return (return_value);
}
```

The subroutines above also use time_UTC_normalize.

```
/* Make a modified tm structure have all of its elements in range.
 * We carry the tm_sec field separately as a long long int so
 * the caller can make large adjustments to the field.  */

int
time_UTC_normalize (struct tm *time_tm, long long int in_seconds,
                    int variable_length_seconds_before_1972)
{
  int return_value = 0;
  long long int seconds;
  struct tm local_tm;

  seconds = in_seconds;

  /* The main body of time_UTC_normalize is an infinite loop so we
   * can say "continue" to restart the algorithm.  The loop
   * terminates only when all of the tests have passed without
   * changing any of the fields.  */
  for (;;)
    {
      /* We test the fields from high-order to low-order, because
       * sometimes the valid range of a field depends on the
       * values of the higher-order fields.  */

      /* Years has no limits.  */

      /* Months are limited to 0 to 11.  If the month is out of
       * range, adjust it and compensate by adjusting the year.  */
      if (time_tm->tm_mon < 0)
        {
          time_tm->tm_mon = time_tm->tm_mon + 12;
          time_tm->tm_year = time_tm->tm_year - 1;
          continue;
        }

      if (time_tm->tm_mon > 11)
```

```
  {
    time_tm->tm_mon = time_tm->tm_mon - 12;
    time_tm->tm_year = time_tm->tm_year + 1;
    continue;
  }

/* The valid range of the day of the month depends
 * on the month and the year.  We walk forward or back
 * through the months one at a time because we need to
 * measure their lengths.  */
if (time_tm->tm_mday < 1)
  {
    time_tm->tm_mday = time_tm->tm_mday +
      time_length_prev_month (time_tm);
    time_tm->tm_mon = time_tm->tm_mon - 1;
    continue;
  }

if (time_tm->tm_mday > time_length_month (time_tm))
  {
    time_tm->tm_mday =
      time_tm->tm_mday - time_length_month (time_tm);
    time_tm->tm_mon = time_tm->tm_mon + 1;
    continue;
  }

/* Hours.  There are always 24 hours in a day.  */
if (time_tm->tm_hour < 0)
  {
    time_tm->tm_hour = time_tm->tm_hour + 24;
    time_tm->tm_mday = time_tm->tm_mday - 1;
    continue;
  }

if (time_tm->tm_hour > 23)
  {
    time_tm->tm_hour = time_tm->tm_hour - 24;
    time_tm->tm_mday = time_tm->tm_mday + 1;
    continue;
  }

/* Minute.  There are always 60 minutes in an hour.  */
if (time_tm->tm_min < 0)
  {
    time_tm->tm_min = time_tm->tm_min + 60;
    time_tm->tm_hour = time_tm->tm_hour - 1;
```

```
        continue;
      }

    if (time_tm->tm_min > 59)
      {
        time_tm->tm_min = time_tm->tm_min - 60;
        time_tm->tm_hour = time_tm->tm_hour + 1;
        continue;
      }

    /* Seconds.  There are 59, 60 or 61 seconds in a minute,
     * depending on the year, month, day, hour and minute.
     * All of those fields are now in their valid range,
     * so we can compute the number of seconds in this
     * minute.  Note that seconds is not carried in the tm
     * structure but passed in separately, so it can be a
     * long long int.  When time_UTC_normalize completes,
     * tm_sec has been set to a valid value.  */
    if (seconds < 0)
      {
        seconds = seconds +
          time_length_prev_UTC_minute (time_tm,
                           variable_length_seconds_before_1972);
        time_tm->tm_min = time_tm->tm_min - 1;
        continue;
      }

    if (seconds >=
        time_length_UTC_minute (time_tm,
                            variable_length_seconds_before_1972))
      {
        seconds = seconds -
          time_length_UTC_minute (time_tm,
                             variable_length_seconds_before_1972);
        time_tm->tm_min = time_tm->tm_min + 1;
        continue;
      }

    break;
  }

/* Now that the seconds value is in its proper range,
 * we can keep it in the tm structure.  */
time_tm->tm_sec = seconds;

/* Use mktime to set the tm_yday and tm_wday fields.
```

```
   * Round down to the nearest minute
   * before calling mktime, so it won't see a leap second.
   */
  time_copy_tm (time_tm, &local_tm);
  local_tm.tm_sec = 0;
  mktime (&local_tm);

  /* Copy back the information returned by mktime.  */
  time_tm->tm_yday = local_tm.tm_yday;
  time_tm->tm_wday = local_tm.tm_wday;

  return (return_value);

}
```

5.3 Doing the same with local time

If you have a time value stored in a tm structure, and it represents local time, you can add to the year, month, day, hour, minute or second field. You can decrease these fields by adding a negative value. When the target date is invalid, such as February 31, you can choose whether to round towards the future or towards the past to get a valid date.

```
/* Add to a field of a tm structure, and then make all the fields
 * have valid values.  */

/* Rounding Mode is used when a modification to a field has caused
 * another field to be invalid.  For example, if you start with
 * October 31 and increase the month by one, you have November 31,
 * which does not exist.  If Rounding Mode is -1, the result is
 * November 30.  If Rounding Mode is +1, the result is December 1.
 * Another example is birthdays on February 29.  If you were born
 * on February 29, 1996, your twentieth birthday is on February 29,
 * 2016.  What day is your twenty-first birthday?  Start with
 * February 29, 1996 and increase the year by twenty-one
 * to get February 29, 2017, then set Rounding Mode to +1
 * if you want March 1, 2017, or -1 if you want February 28, 2017.
 *
 * Like months, not all minutes are the same length.  If you start
 * with December 31, 2016, 18:59:60 EST, and decrease the minutes
 * by one you get December 31, 2016, 18:58:60 EST, which does not
 * exist.  If Rounding Mode is +1 you get December 31, 2016,
 * 18:59:00 EST.  If Rounding Mode is -1 you get December 31, 2016,
 * 18:58:59 EST.
 */
```

```
/* After changing the day, make sure the seconds are correct.  */
static int
time_local_adjust_seconds (struct tm *time_tm, int rounding_mode,
                           int variable_length_seconds_before_1972)
{

  /* The seconds can be invalid if we are at the end of the UTC day
   * and the day we came from was longer than this day.  This can
   * happen if the day we came from was 86,401 seconds long, or if
   * the day we came to was 86,399 seconds long, or both.  */

  /* Adjust the local second, if necessary.  The adjustment may
   * propagate to higher-order fields.  Note that leap seconds
   * happen at the same time everywhere, regardless of time zone.  */
  if (time_tm->tm_sec >=
      time_length_local_minute (time_tm,
                                variable_length_seconds_before_1972))
    {
      switch (rounding_mode)
        {
        case 1:
          /* Round up to the first second of the next minute.  */
          time_tm->tm_sec = 0;
          time_tm->tm_min = time_tm->tm_min + 1;
          if (time_tm->tm_min > 59)
            {
              time_tm->tm_min = 0;
              time_tm->tm_hour = time_tm->tm_hour + 1;
              if (time_tm->tm_hour > 23)
                {
                  time_tm->tm_hour = 0;
                  time_tm->tm_mday = time_tm->tm_mday + 1;
                  if (time_tm->tm_mday > time_length_month (time_tm))
                    {
                      time_tm->tm_mday = 1;
                      time_tm->tm_mon = time_tm->tm_mon + 1;
                      if (time_tm->tm_mon > 11)
                        {
                          time_tm->tm_mon = 0;
                          time_tm->tm_year = time_tm->tm_year + 1;
                        }
                    }
                }
            }
          break;
```

```
        case -1:
          /* Round down to the last second of the day.  */
          time_tm->tm_sec =
            time_length_local_minute (time_tm,
                                      variable_length_seconds_before_1972)
            - 1;
          break;

        default:
          break;
        }
    }

  return (0);
}

/* Add a specified number of years to a specified time.  */
int
time_local_add_years (struct tm *time_tm, int addend, int rounding_mode,
                      int variable_length_seconds_before_1972)
{
  int return_value;

  time_tm->tm_year = time_tm->tm_year + addend;

  /* We have just changed the year.  If the day of the month is outside
   * its valid range, which can only be the case for February 29, round it
   * up to March 1 or down to February 28, depending on the rounding mode.
   */
  if (time_tm->tm_mday > time_length_month (time_tm))
    {
      switch (rounding_mode)
        {
        case 1:
          /* Round up to March 1.  */
          time_tm->tm_mday = 1;
          time_tm->tm_mon = time_tm->tm_mon + 1;
          while (time_tm->tm_mon > 11)
            {
              time_tm->tm_mon = time_tm->tm_mon - 12;
              time_tm->tm_year = time_tm->tm_year + 1;
            }
          break;

        case -1:
          /* Round down to February 28.  */
```

```
          time_tm->tm_mday = time_length_month (time_tm);
          break;

        default:
          break;
        }
    }

  /* Adjust the value of the seconds field to be valid.  */
  return_value =
    time_local_adjust_seconds (time_tm, rounding_mode,
                               variable_length_seconds_before_1972);
  if (return_value != 0)
    return (return_value);

  return_value =
    time_local_normalize (time_tm, time_tm->tm_sec,
                          variable_length_seconds_before_1972);
  return (return_value);
}

/* Add a specified number of months to the time.  To subtract months,
 * make the amount to add negative.  */
int
time_local_add_months (struct tm *time_tm, int addend, int rounding_mode,
                       int variable_length_seconds_before_1972)
{
  int return_value;

  time_tm->tm_mon = time_tm->tm_mon + addend;

  /* Get the months value into its valid range, compensating by
   * adjusting the years.  */
  while (time_tm->tm_mon < 0)
    {
      time_tm->tm_mon = time_tm->tm_mon + 12;
      time_tm->tm_year = time_tm->tm_year - 1;
    }
  while (time_tm->tm_mon > 11)
    {
      time_tm->tm_mon = time_tm->tm_mon - 12;
      time_tm->tm_year = time_tm->tm_year + 1;
    }

  /* If we are at the end of the month, we may have to adjust the day
   * to the last day of this month or the first day of the next.  */
```

```
  if (time_tm->tm_mday > time_length_month (time_tm))
    {
      switch (rounding_mode)
        {
        case 1:
          time_tm->tm_mday = 1;
          time_tm->tm_mon = time_tm->tm_mon + 1;
          if (time_tm->tm_mon > 11)
            {
              /* This is only a formality--December has 31 days
               * so we will never have to round up to January of
               * the next year.  */
              time_tm->tm_mon = time_tm->tm_mon - 12;
              time_tm->tm_year = time_tm->tm_year + 1;
            }
          break;

        case -1:
          time_tm->tm_mday = time_length_month (time_tm);
          break;

        default:
          break;
        }
    }

  /* Make sure the value of the seconds field is valid.  */
  return_value =
    time_local_adjust_seconds (time_tm, rounding_mode,
                               variable_length_seconds_before_1972);
  if (return_value != 0)
    return (return_value);

  return_value =
    time_local_normalize (time_tm, time_tm->tm_sec,
                          variable_length_seconds_before_1972);
  return (return_value);
}

/* Add a specified number of days to a specified time.  */
int
time_local_add_days (struct tm *time_tm, int addend, int rounding_mode,
                     int variable_length_seconds_before_1972)
{
  int return_value;
```

```
  time_tm->tm_mday = time_tm->tm_mday + addend;

  /* Go through the months one at a time because they have
   * non-uniform lengths.  */
  while (time_tm->tm_mday < 1)
    {
      /* We are backing into the previous month.  */
      time_tm->tm_mon = time_tm->tm_mon - 1;
      if (time_tm->tm_mon < 0)
        {
          time_tm->tm_mon = time_tm->tm_mon + 12;
          time_tm->tm_year = time_tm->tm_year - 1;
        }
      time_tm->tm_mday =
        time_tm->tm_mday + time_length_month (time_tm);
    }

  while (time_tm->tm_mday > time_length_month (time_tm))
    {
      /* We are moving forward into future months.  */
      time_tm->tm_mday =
        time_tm->tm_mday - time_length_month (time_tm);
      time_tm->tm_mon = time_tm->tm_mon + 1;
      if (time_tm->tm_mon > 11)
        {
          time_tm->tm_mon = time_tm->tm_mon - 12;
          time_tm->tm_year = time_tm->tm_year + 1;
        }
    }

  /* Make sure the value of the seconds field is valid.  */
  return_value =
    time_local_adjust_seconds (time_tm, rounding_mode,
                               variable_length_seconds_before_1972);
  if (return_value != 0)
    return (return_value);

  return_value =
    time_local_normalize (time_tm, time_tm->tm_sec,
                          variable_length_seconds_before_1972);
  return (return_value);

}

/* Add a specified number of hours to the specified time.  */
int
```

```
time_local_add_hours (struct tm *time_tm, int addend,
                      int rounding_mode,
                      int variable_length_seconds_before_1972)
{
  int return_value;

  time_tm->tm_hour = time_tm->tm_hour + addend;

  /* If the hours field is out of range, adjust it.  */
  while (time_tm->tm_hour < 0)
    {
      time_tm->tm_hour = time_tm->tm_hour + 24;
      time_tm->tm_mday = time_tm->tm_mday - 1;
      if (time_tm->tm_mday < 1)
        {
          time_tm->tm_mday =
            time_tm->tm_mday + time_length_prev_month (time_tm);
          time_tm->tm_mon = time_tm->tm_mon - 1;
          if (time_tm->tm_mon < 0)
            {
              time_tm->tm_mon = time_tm->tm_mon + 12;
              time_tm->tm_year = time_tm->tm_year - 1;
            }
        }
    }

  while (time_tm->tm_hour > 23)
    {
      time_tm->tm_hour = time_tm->tm_hour - 24;
      time_tm->tm_mday = time_tm->tm_mday + 1;
      if (time_tm->tm_mday > time_length_month (time_tm))
        {
          time_tm->tm_mday =
            time_tm->tm_mday - time_length_month (time_tm);
          time_tm->tm_mon = time_tm->tm_mon + 1;
          if (time_tm->tm_mon > 11)
            {
              time_tm->tm_mon = time_tm->tm_mon - 12;
              time_tm->tm_year = time_tm->tm_year + 1;
            }
        }
    }

  /* Make sure the seconds field is valid.  */
  return_value =
    time_local_adjust_seconds (time_tm, rounding_mode,
```

```
                                variable_length_seconds_before_1972);
  if (return_value != 0)
    return (return_value);

  return_value =
    time_local_normalize (time_tm, time_tm->tm_sec,
                          variable_length_seconds_before_1972);
  return (return_value);
}

/* Add a specified number of minutes to the specified time.  */
int
time_local_add_minutes (struct tm *time_tm, int addend,
                        int rounding_mode,
                        int variable_length_seconds_before_1972)
{
  int return_value;

  time_tm->tm_min = time_tm->tm_min + addend;

  /* If the minutes field is out of range, adjust it.  */
  while (time_tm->tm_min < 0)
    {
      time_tm->tm_min = time_tm->tm_min + 60;
      time_tm->tm_hour = time_tm->tm_hour - 1;
      if (time_tm->tm_hour < 0)
        {
          time_tm->tm_hour = time_tm->tm_hour + 24;
          time_tm->tm_mday = time_tm->tm_mday - 1;
          if (time_tm->tm_mday < 1)
            {
              time_tm->tm_mday =
                time_tm->tm_mday + time_length_prev_month (time_tm);
              time_tm->tm_mon = time_tm->tm_mon - 1;
              if (time_tm->tm_mon < 0)
                {
                  time_tm->tm_mon = time_tm->tm_mon + 12;
                  time_tm->tm_year = time_tm->tm_year - 1;
                }
            }
        }
    }

  while (time_tm->tm_min > 59)
    {
      time_tm->tm_min = time_tm->tm_min - 60;
```

```
      time_tm->tm_hour = time_tm->tm_hour + 1;
      if (time_tm->tm_hour > 23)
        {
          time_tm->tm_hour = time_tm->tm_hour - 24;
          time_tm->tm_mday = time_tm->tm_mday + 1;
          if (time_tm->tm_mday > time_length_month (time_tm))
            {
              time_tm->tm_mday =
                time_tm->tm_mday - time_length_month (time_tm);
              time_tm->tm_mon = time_tm->tm_mon + 1;
              if (time_tm->tm_mon > 11)
                {
                  time_tm->tm_mon = time_tm->tm_mon - 12;
                  time_tm->tm_year = time_tm->tm_year + 1;
                }
            }
        }
    }

  /* Make sure the seconds field is valid.  */
  return_value =
    time_local_adjust_seconds (time_tm, rounding_mode,
                               variable_length_seconds_before_1972);
  if (return_value != 0)
    return (return_value);

  return_value =
    time_local_normalize (time_tm, time_tm->tm_sec,
                          variable_length_seconds_before_1972);
  return (return_value);
}

/* Add a specified number of seconds to a time.  */
int
time_local_add_seconds (struct tm *time_tm, long long int add_seconds,
                        int variable_length_seconds_before_1972)
{
  long long int add_nanoseconds;
  long long int nanoseconds;
  int return_val;

  /* use time_local_add_seconds_ns with nanoseconds == 0.  */
  add_nanoseconds = 0;
  nanoseconds = 0;
  return_val =
    time_local_add_seconds_ns (time_tm, &nanoseconds,
```

```
                                add_seconds, add_nanoseconds,
                                variable_length_seconds_before_1972);
  return (return_val);
}

/* Add a specified number of seconds and nanoseconds to a time.  */
int
time_local_add_seconds_ns (struct tm *time_tm,
                           long long int *nanoseconds,
                           long long int add_seconds,
                           long long int add_nanoseconds,
                           int variable_length_seconds_before_1972)
{
  long long int accumulated_seconds;
  long long int accumulated_nanoseconds;
  int return_value;

  accumulated_seconds = time_tm->tm_sec + add_seconds;
  accumulated_nanoseconds = *nanoseconds + add_nanoseconds;
  while (accumulated_nanoseconds >= 1e9)
    {
      accumulated_seconds = accumulated_seconds + 1;
      accumulated_nanoseconds = accumulated_nanoseconds - 1e9;
    }
  while (accumulated_nanoseconds < 0)
    {
      accumulated_seconds = accumulated_seconds - 1;
      accumulated_nanoseconds = accumulated_nanoseconds + 1e9;
    }

  /* The caller can increase or decrease the seconds by a large
   * amount to navigate the calendar by seconds.  Adjust the year,
   * month, day of the month, hour and minute fields to get the
   * seconds field into its valid range, and thereby name the
   * signated second using the Gregorian calendar.  Note that
   * time_local_normalize takes the seconds value in a
   * long long int parameter and ignores the tm_sec field of time_tm.
   * This is to allow the number of seconds to be very large or
   * very negative.  When time_local_normalize is complete it
   * stores the adjusted value of the seconds parameter into
   * time_tm->tm_sec.  */
  return_value =
    time_local_normalize (time_tm, accumulated_seconds,
                          variable_length_seconds_before_1972);
  *nanoseconds = accumulated_nanoseconds;
  return (return_value);
```

```
}
```

The subroutines above use time_local_normalize.

```
/* Make a modified tm structure containing local time have all of
 * its elements in range.  We carry the tm_sec field separately as
 * a long long int so the caller can make large adjustments to the
 * field.  */
int
time_local_normalize (struct tm *time_tm, long long int in_seconds,
                      int variable_length_seconds_before_1972)
{
  int return_value = 0;
  long long int seconds;
  struct tm local_tm;

  seconds = in_seconds;

  /* The body of time_local_normalize is an infinite loop so we can
   * say "continue" to restart the algorithm.  The loop terminates
   * only when all of the tests have passed without manipulating
   * any of the fields.  */
  for (;;)
    {
      /* We test the fields from high-order to low-order, because
       * sometimes the valid range of a field depends on the values
       * of the higher-order fields.  */

      /* Years has no limits.  */

      /* Months are limited to 0 to 11.  If the month is
       * out of range, adjust it and compensate by adjusting
       * the year.  */
      if (time_tm->tm_mon < 0)
        {
          time_tm->tm_mon = time_tm->tm_mon + 12;
          time_tm->tm_year = time_tm->tm_year - 1;
          continue;
        }

      if (time_tm->tm_mon > 11)
        {
          time_tm->tm_mon = time_tm->tm_mon - 12;
          time_tm->tm_year = time_tm->tm_year + 1;
          continue;
        }
```

```
/* The valid range of the day of the month depends
 * on the month and the year.  We walk forward or back
 * through the months one at a time because we need to
 * measure their lengths.  */
if (time_tm->tm_mday < 1)
  {
    time_tm->tm_mday = time_tm->tm_mday +
      time_length_prev_month (time_tm);
    time_tm->tm_mon = time_tm->tm_mon - 1;
    continue;
  }

if (time_tm->tm_mday > time_length_month (time_tm))
  {
    time_tm->tm_mday =
      time_tm->tm_mday - time_length_month (time_tm);
    time_tm->tm_mon = time_tm->tm_mon + 1;
    continue;
  }

/* Hours.  There are always 24 hours in a day.  */
if (time_tm->tm_hour < 0)
  {
    time_tm->tm_hour = time_tm->tm_hour + 24;
    time_tm->tm_mday = time_tm->tm_mday - 1;
    continue;
  }

if (time_tm->tm_hour > 23)
  {
    time_tm->tm_hour = time_tm->tm_hour - 24;
    time_tm->tm_mday = time_tm->tm_mday + 1;
    continue;
  }

/* Minute.  There are always 60 minutes in an hour.  */
if (time_tm->tm_min < 0)
  {
    time_tm->tm_min = time_tm->tm_min + 60;
    time_tm->tm_hour = time_tm->tm_hour - 1;
    continue;
  }

if (time_tm->tm_min > 59)
  {
    time_tm->tm_min = time_tm->tm_min - 60;
```

```
          time_tm->tm_hour = time_tm->tm_hour + 1;
          continue;
        }

      /* Seconds.  There are 59, 60 or 61 seconds in a minute,
       * depending on the year, month, day, hour and minute.
       * All of those fields are now in their valid range,
       * so we can compute the number of seconds in this
       * minute.  Note that seconds is not carried in the tm
       * structure but passed in separately, so it can be a
       * long long int.  When time_local_normalize completes,
       * tm_sec has been set to a valid value.  */
      if (seconds < 0)
        {
          seconds = seconds +
            time_length_prev_local_minute (time_tm,
                           variable_length_seconds_before_1972);
          time_tm->tm_min = time_tm->tm_min - 1;
          continue;
        }

      if (seconds >=
          time_length_local_minute (time_tm,
                            variable_length_seconds_before_1972))
        {
          seconds = seconds -
            time_length_local_minute (time_tm,
                            variable_length_seconds_before_1972);
          time_tm->tm_min = time_tm->tm_min + 1;
          continue;
        }
      break;
    }

  /* Now that the seconds value is in its proper range
   * we can store it in the tm structure.  */
  time_tm->tm_sec = seconds;

  /* Use mktime to compute the other fields, given
   * tm_year, tm_mon, tm_mday, tm_hour, tm_min,
   * tm_sec and tm_gmtoff.  We set the seconds to 0
   * so that mktime won't be presented with a leap
   * second.  */
  time_copy_tm (time_tm, &local_tm);
  local_tm.tm_sec = 0;
  mktime (&local_tm);
```

```
  /* Copy back the information returned by mktime.
   * If the time zone changed, the hour likely changed,
   * and possibly the year, month, day and minute.
   * Daylight Saving Time always occurs on minute
   * boundaries, so the seconds won't have changed.  */
  time_tm->tm_year = local_tm.tm_year;
  time_tm->tm_mon = local_tm.tm_mon;
  time_tm->tm_mday = local_tm.tm_mday;
  time_tm->tm_hour = local_tm.tm_hour;
  time_tm->tm_min = local_tm.tm_min;
  time_tm->tm_yday = local_tm.tm_yday;
  time_tm->tm_wday = local_tm.tm_wday;
  time_tm->tm_isdst = local_tm.tm_isdst;
  time_tm->tm_zone = local_tm.tm_zone;
  time_tm->tm_gmtoff = local_tm.tm_gmtoff;

  return (return_value);
}
```

5.4 Representing a time as a string

If you want to send a time to another computer, or write it into a file, it is convenient to be able to convert it to a string. Here is how you can do that. This subroutine works for both Coordinated Universal Time and local time.

```
/* Subroutine to convert a time in tm format to a string,
 * in RFC 3339 (ISO 8601) format.  Returns the number of
 * characters written into the string, not counting the
 * trailing NUL byte.  If the time zone is not an integral
 * number of minutes offset from UTC, violates RFC 3339
 * by appending ":ss" to the offset.  */
int
time_tm_to_string (struct tm *input_tm,
                   char *current_time_string,
                   int current_time_string_length)
{
  int return_val;
  int gmt_offset, offset_hours, offset_minutes, offset_seconds;
  int pos_offset;
  char *timezone;
  char buffer [32];

  /* Compute the time zone information. We use Z for UTC,
   * otherwise + or - hh:mm or hh:mm:ss.  */
  gmt_offset = input_tm->tm_gmtoff;
```

```
  if (gmt_offset == 0)
    {
      timezone = "Z";
    }
  else
    {
      if (gmt_offset < 0)
        {
          pos_offset = -gmt_offset;
        }
      else
        {
          pos_offset = gmt_offset;
        }

      offset_hours = pos_offset / 3600;
      offset_minutes = ((pos_offset / 60) - (offset_hours * 60));
      offset_seconds =
        pos_offset - (offset_hours * 3600) - (offset_minutes * 60);
      if (gmt_offset < 0)
        {
          offset_hours = -offset_hours;
        }

      if (offset_seconds == 0)
        {
          snprintf (&buffer [0], sizeof (buffer),
                    "%+03d:%02d", offset_hours, offset_minutes);
        }
      else
        {
          snprintf (&buffer [0], sizeof (buffer),
                    "%+03d:%02d:%02d", offset_hours, offset_minutes,
                    offset_seconds);
        }
      timezone = &buffer [0];
    }

  return_val = snprintf (current_time_string,
                         current_time_string_length,
                         "%04d-%02d-%02dT%02d:%02d:%02d%s",
                         input_tm->tm_year + 1900,
                         input_tm->tm_mon + 1,
                         input_tm->tm_mday,
                         input_tm->tm_hour,
                         input_tm->tm_min,
```

```
                        input_tm->tm_sec,
                        timezone);

  return (return_val);
}
```

If you need more precision you can include the nanoseconds:

```
/* Subroutine to convert a time in tm format plus the number
 * of nanoseconds since the last second to a string,
 * in RFC 3339 (ISO 8601) format.  Returns the number of
 * characters written into the string, not counting the
 * trailing NUL byte.  If the time zone is not an integral
 * number of minutes offset from UTC, violates RFC 3339
 * by appending ":ss" to the offset.  */
int
time_tm_nano_to_string (struct tm *input_tm,
                        int input_nanoseconds,
                        char *current_time_string,
                        int current_time_string_length)
{
  int return_val;
  int gmt_offset, offset_hours, offset_minutes, offset_seconds;
  int pos_offset;
  char *timezone;
  char buffer [32];

  /* Compute the time zone information. We use Z for UTC,
   * otherwise + or - hh:mm or hh:mm:ss.  */
  gmt_offset = input_tm->tm_gmtoff;
  if (gmt_offset == 0)
    {
      timezone = "Z";
    }
  else
    {
      if (gmt_offset < 0)
        {
          pos_offset = -gmt_offset;
        }
      else
        {
          pos_offset = gmt_offset;
        }

      offset_hours = pos_offset / 3600;
      offset_minutes = ((pos_offset / 60) - (offset_hours * 60));
```

```
      offset_seconds =
        pos_offset - (offset_hours * 3600) - (offset_minutes * 60);
      if (gmt_offset < 0)
        {
          offset_hours = -offset_hours;
        }

      if (offset_seconds == 0)
        {
          snprintf (&buffer [0], sizeof (buffer),
                    "%+03d:%02d", offset_hours, offset_minutes);
        }
      else
        {
          snprintf (&buffer [0], sizeof (buffer),
                    "%+03d:%02d:%02d", offset_hours, offset_minutes,
                    offset_seconds);
        }
      timezone = &buffer [0];
    }

  return_val = snprintf (current_time_string,
                         current_time_string_length,
                         "%04d-%02d-%02dT%02d:%02d:%02d.%09d%s",
                         input_tm->tm_year + 1900,
                         input_tm->tm_mon + 1,
                         input_tm->tm_mday,
                         input_tm->tm_hour,
                         input_tm->tm_min,
                         input_tm->tm_sec,
                         input_nanoseconds,
                         timezone);

  return (return_val);
}
```

6 List of Entry Points

For reference, here are the subroutines described above, in alphabetical order, each with a brief description.

```
/* Convert a 128-bit integer to a string.  */
int
int128_to_string (__int128 value,
                  char *result, int result_size);
```

```c
/* Copy a tm structure.  */
int
time_copy_tm (struct tm *in_tm, struct tm *out_tm);

/* Fetch the current time, without nanoseconds.  */
int
time_current_tm (struct tm *current_tm);

/* Fetch the current time, including nanoseconds.  */
int
time_current_tm_nano (struct tm *current_tm,
                      int *nanoseconds);

/* Compute the difference between two times in seconds.  */
long long int
time_diff (struct tm *A_tm, struct tm *B_tm,
           int variable_length_seconds_before_1972)
  __attribute__ ((pure));

/* Fetch the difference between TAI and UTC at the beginning
 * of the specified Julian Day Number.  */
int
time_DTAI (int Julian_day_number,
           int variable_length_seconds_before_1972)
  __attribute__ ((const));

/* Compute the Julian Day Number corresponding to a date.  */
int
time_Julian_day_number (int year, int month, int day)
  __attribute__ ((const));

/* Compute the length of the current minute of local time.
 */
int
time_length_local_minute (struct tm *time_tm,
                          int variable_length_seconds_before_1972);

/* Compute the length of the current month.  */
int
time_length_month (struct tm *time_tm)
  __attribute__ ((pure));

/* Compute the length of the previous minute of local time.
 */
int
```

```
time_length_prev_local_minute (struct tm *time_tm,
                               int variable_length_seconds_before_1972);

/* Compute the length of the previous month.  */
int
time_length_prev_month (struct tm *time_tm);

/* Compute the length of the previous minute of
 * Coordinated Universal Time.  */
int
time_length_prev_UTC_minute (struct tm *time_tm,
                             int variable_length_seconds_before_1972);

/* Compute the length of the current minute of
 * Coordinated Universal Time.  */
int
time_length_UTC_minute (struct tm *time_tm,
                        int variable_length_seconds_before_1972)
  __attribute__ ((pure));

/* Add days to a local time.  */
int
time_local_add_days (struct tm *time_tm, int addend,
                     int rounding_mode,
                     int variable_length_seconds_before_1972);

/* Add hours to a local time.  */
int
time_local_add_hours (struct tm *time_tm, int addend,
                      int rounding_mode,
                      int variable_length_seconds_before_1972);

/* Add minutes to a local time.  */
int
time_local_add_minutes (struct tm *time_tm, int addend,
                        int rounding_mode,
                        int variable_length_seconds_before_1972);

/* Add months to a local time.  */
int
time_local_add_months (struct tm *time_tm, int addend,
                       int rounding_mode,
                       int variable_length_seconds_before_1972);

/* Add seconds to a local time.  */
int
```

```
time_local_add_seconds (struct tm *time_tm,
                        long long int add_seconds,
                        int variable_length_seconds_before_1972);

/* Add seconds and nanoseconds to a local time.  */
int
time_local_add_seconds_ns (struct tm *time_tm,
                           long long int *nanoseconds,
                           long long int add_seconds,
                           long long int add_nanoseconds,
                           int variable_length_seconds_before_1972);

/* Add years to a local time.  */
int
time_local_add_years (struct tm *time_tm, int addend,
                      int rounding_mode,
                      int variable_length_seconds_before_1972);

/* Make sure all of the fields of a tm structure containing
 * local time are within their valid ranges.  */
int
time_local_normalize (struct tm *time_tm,
                      long long int seconds,
                      int variable_length_seconds_before_1972);

/* Convert local time to Coordinated Universal Time.  */
int
time_local_to_UTC (struct tm *local_time,
                   struct tm *coordinated_universal_time,
                   int variable_length_seconds_before_1972);

/* Sleep until a specified Coordinated Universal Time.  */
int
time_sleep_until (struct tm *time_tm, int nanoseconds,
                  int variable_length_seconds_before_1972);

/* Convert the time and nanoseconds to a 128-bit integer.
 */
int
time_tm_nano_to_integer (struct tm *input_tm,
                         int input_nanoseconds,
                         __int128 *result);

/* Convert the time and nanoseconds to a string.  */
int
time_tm_nano_to_string (struct tm *input_tm,
```

```
                        int input_nanoseconds,
                        char *current_time_string,
                        int current_time_string_length);

/* Convert the time to a long long integer.  */
int
time_tm_to_integer (struct tm *input_tm,
                    long long int *result);

/* Convert the time to a string.  */
int
time_tm_to_string (struct tm *input_tm,
                   char *current_time_string,
                   int current_time_string_length);

/* Add days to a Coordinated Universal Time.  */
int
time_UTC_add_days (struct tm *time_tm, int addend,
                   int rounding_mode,
                   int variable_length_seconds_before_1972);

/* Add hours to a Coordinated Universal Time.  */
int
time_UTC_add_hours (struct tm *time_tm, int addend,
                    int rounding_mode,
                    int variable_length_seconds_after_1972);

/* Add minutes to a Coordinated Universal Time.  */
int
time_UTC_add_minutes (struct tm *time_tm, int addend,
                      int rounding_mode,
                      int variable_length_seconds_before_1972);

/* Add months to a Coordinated Universal Time.  */
int
time_UTC_add_months (struct tm *time_tm, int addend,
                     int rounding_mode,
                     int variable_length_seconds_before_1972);

/* Add seconds to a Coordinated Universal Time.  */
int
time_UTC_add_seconds (struct tm *time_tm,
                      long long int add_seconds,
                      int variable_length_seconds_before_1972);

/* Add seconds and nanoseconds to a Coordinated Universal Time.
```

```
 */
int
time_UTC_add_seconds_ns (struct tm *time_tm,
```

7 Embedded files

To save you the trouble of typing in these subroutines, they are embedded in this PDF file. You can extract them using okular or Adobe Acrobat Reader. Also included are some sample and test programs, and a Makefile to compile the programs and construct this PDF file.

For the benefit of readers who are unable to extract the software, Steve Summit has kindly hosted the files on his web site, at this URL: `http://www.eskimo.com/~scs/time/sauter`.

8 Examples

These examples are intended to illustrate how an application would use these subroutines to manipulate time represented in a tm structure.

8.1 Meeting at 9 AM

You wish to schedule a meeting in the Chrysanthemum Conference Hall at 9 AM on June 30, 2017. The participants are all over the world, so the message sent to their scheduling software will be in Coordinated Universal Time. The Chrysanthemum Conference Hall is located in Tokyo, Japan.

Set your time zone to Asia/Tokyo. Construct a tm structure containing June 30, 2017. Convert that from local time to UTC. Convert the UTC time to a string, and send the string to the participants. The string you send should be 2017-06-30T00:00:00Z.

8.2 Periodic Backups

You are writing the scheduler for a backup program. At 3 AM local time after each weekday it does an incremental backup, except it does a full backup if the last full backup is more than a month old. The scheduler is run when the backup program is installed on the computer, and again when the current backup has started. The scheduler is expected to compute the Coordinated Universal Time at which to start the next backup, and indicate whether the backup will be incremental or full.

Compute the current time and convert it to local time. Set the hour, minute and second to 03:00:00. Using that as the base, advance by one or more days until tm_wday is not Sunday or Monday. Convert the result to Coordinated Universal Time. This is the time of the next backup. If you install the software on December 31, 2016, at 21:21:35 in time zone America/New_York, the first scheduled backup will be at 2017-01-03T08:00:00Z.

```
struct tm time_tm;
struct tm local_time_tm;
struct tm backup_tm;
struct tm extra_time_tm;
struct tm int_time_tm;
char buffer1 [64];
char buffer2 [64];
int found_time, day_incr, diff;

time_current_tm (&time_tm);
time_UTC_to_local (&time_tm, &local_time_tm, 0);
time_tm_to_string (&local_time_tm, &buffer1 [0], sizeof(buffer1));
printf ("now: %s\n", buffer1);
local_time_tm.tm_hour = 3;
local_time_tm.tm_min = 0;
local_time_tm.tm_sec = 0;

found_time = 0;
day_incr = 1;
while (!found_time)
  {
    time_copy_tm (&local_time_tm, &extra_time_tm);
    time_local_add_days (&extra_time_tm, day_incr, 1, 0);
    if ((extra_time_tm.tm_wday == 0) ||
        (extra_time_tm.tm_wday == 1))
      day_incr = day_incr + 1;
    else
      found_time = 1;
  }

time_local_to_UTC (&extra_time_tm, &time_tm, 0);
time_tm_to_string (&time_tm, &buffer1 [0], sizeof (buffer1));
printf ("Next scheduled backup is at %s.\n", buffer1);
```

Subtract one month from the local time computed above, and convert it to Coordinated Universal Time. Compute the difference between this time and the time of the last full backup. If the difference is negative, or if there has never been a full backup, this backup is full; otherwise it is incremental.

```
/* To illustrate, assume the last backup was 29 days ago.
 */
time_current_tm (&backup_tm);
time_UTC_add_days (&backup_tm, -29, -1, 0);
time_tm_to_string (&backup_tm, &buffer2 [0], sizeof (buffer2));
printf ("Assume the last full backup was %s.\n", buffer2);

time_local_add_months (&extra_time_tm, -1, 1, 0);
```

```
time_local_to_UTC (&extra_time_tm, &int_time_tm, 0);
time_tm_to_string (&int_time_tm, &buffer1 [0], sizeof (buffer1));
printf ("One month before the next scheduled backup is %s.\n",
        buffer1);
diff = time_diff (&int_time_tm, &backup_tm, 0);
if (diff < 0)
  printf ("Next backup is full.\n");
else
  printf ("Next backup is incremental.\n");
```

8.3 Rocket over Central Park

You are asked to explode a fireworks rocket over Central Park in New York at exactly 7 PM on the last day of each month. You place a computer-controlled rocket launcher on a building adjacent to Central Park. You know that the rocket will take exactly two seconds to reach the top of its arc after it is launched. The launch computer knows only about Coordinated Universal Time. What schedule do you give it for launching in 2016?

Starting with January 31, 2016, in time zone America/New_York, step through the months until the year is no longer 2016, rounding the date down to get the last day of the month. On each day, set the time to 7 PM, convert to UTC, then subtract two seconds. The schedule will accomodate Daylight Saving Time and the leap second at the end of the year.

```
struct tm time_tm;
struct tm local_time_tm;
struct tm extra_time_tm;
char buffer1 [64];
char buffer2 [64];
int month_incr;

time_current_tm (&time_tm);
time_UTC_to_local (&time_tm, &local_time_tm, 0);

local_time_tm.tm_year = 2016 - 1900;
local_time_tm.tm_mon = 1 - 1;
local_time_tm.tm_mday = 31;
local_time_tm.tm_hour = 19;
local_time_tm.tm_min = 0;
local_time_tm.tm_sec = 0;

month_incr = 0;

while (1)
  {
    time_copy_tm (&local_time_tm, &extra_time_tm);
```

```
    time_local_add_months (&extra_time_tm, month_incr, -1, 0);
    extra_time_tm.tm_hour = 19;
    extra_time_tm.tm_min = 0;
    extra_time_tm.tm_sec = 0;
    if (extra_time_tm.tm_year != (2016 - 1900))
      break;

    time_local_to_UTC (&extra_time_tm, &time_tm, 0);
    time_UTC_add_seconds (&time_tm, -2, 0);
    time_tm_to_string (&time_tm, &buffer1 [0], sizeof (buffer1));
    time_tm_to_string (&extra_time_tm, &buffer2 [0],
                       sizeof (buffer2));
    printf ("%s, 2 sec before %s.\n", buffer1, buffer2);
    month_incr = month_incr + 1;
  }
```

Here is the schedule for 2016:

```
2016-01-31T23:59:58Z, 2 sec before 2016-01-31T19:00:00-05:00.
2016-02-29T23:59:58Z, 2 sec before 2016-02-29T19:00:00-05:00.
2016-03-31T22:59:58Z, 2 sec before 2016-03-31T19:00:00-04:00.
2016-04-30T22:59:58Z, 2 sec before 2016-04-30T19:00:00-04:00.
2016-05-31T22:59:58Z, 2 sec before 2016-05-31T19:00:00-04:00.
2016-06-30T22:59:58Z, 2 sec before 2016-06-30T19:00:00-04:00.
2016-07-31T22:59:58Z, 2 sec before 2016-07-31T19:00:00-04:00.
2016-08-31T22:59:58Z, 2 sec before 2016-08-31T19:00:00-04:00.
2016-09-30T22:59:58Z, 2 sec before 2016-09-30T19:00:00-04:00.
2016-10-31T22:59:58Z, 2 sec before 2016-10-31T19:00:00-04:00.
2016-11-30T23:59:58Z, 2 sec before 2016-11-30T19:00:00-05:00.
2016-12-31T23:59:59Z, 2 sec before 2016-12-31T19:00:00-05:00.
```

9 Limitations

9.1 Unpredictable Rotation of the Earth

The rotation rate of the Earth is not predictable far in advance. It must be observed, like the weather. To say this another way, we do not know exactly when the Sun will rise a year from now. Because Coordinated Universal Time is tied to the rotation of the Earth, it is similarly uncertain.

We have good records of the rotation rate of the Earth from the time of the invention of the telescope and the pendulum clock until the present. Prior to 1700, the measurements become uncertain. Efforts have been made to improve our knowledge of the rotation rate of the Earth in historical times[3][4][7][6][2][8]. This software carries a table in file `dtai_table.tab` that reaches back to the year −2000 and forward to the year 2500 based on the latest research. Beyond the table, the software makes predictions which become less reliable the further out you go.

The table will be updated from time to time to capture present observations of the Earth's rotation and better estimates from historical times. If your application deals with times before 1700 or more than six months in the future, you should recompute your times whenever the DTAI table is updated. In the example of the rocket over Central Park, if you had pre-computed the times in 2015, you would have missed the leap second at the end of December 31, 2016. The update to the DTAI table in the middle of 2016 should cause the schedule for the remainder of the year to be recomputed, resulting in a launch time of 23:59:59 for the December 31 rocket, instead of 23:59:58.

If your application deals only with days, and is not sensitive to the exact number of seconds between two times, you can be more relaxed.

9.2 Only coded for GNU/Linux

The code embedded in this PDF has been written for GNU/Linux. Some effort would be required to port the code to another operating system.

10 Kernel Recommendations

Most of the work needed to handle UTC correctly is done in application programs, but some kernel changes would result in better UTC support.

1. Add a named clock, called `CLOCK_UTC`, which uses a `timeval` in which the nanoseconds field has 1e6 added to indicate a leap second. This would allow the clock to be set during a leap second.

2. Add a kernel boot parameter (perhaps in conjunction with an NTP configuration option) to specify the behavior of `CLOCK_REALTIME`. Choices are:

   (a) The same as the old behavior, where during a leap second 23:59:59 is repeated and `adjtimex` returns `TIME_OOP`. This is the default, for compatibility.

   (b) Smeared time, where the clock begins to depart from UTC several hours before the leap second, reaches its maximum at the leap second, then returns to UTC over the next several hours. `Adjtimex` never returns `TIME_OOP` and no second is repeated. Reasonable choices for the total smear time are 24 and 48 hours. Clock smearing is used to mask leap seconds from software that isn't prepared to handle them.

   (c) The same as `CLOCK_UTC`.

3. When file systems store their mtime, atime and ctime, they use the value returned by `CLOCK_REALTIME`. Provided that `CLOCK_REALTIME` is set to behave like `CLOCK_UTC` this fixes the inaccuracy of file times.

11 Update History

2017-01-27 Update the Makefile and some of the test programs based on feedback from Steve Summit.

2017-01-18 Added the ability to use variable length seconds before 1972.

2017-01-10 Updated to IERS Bulletin C version 53, issued January 9, 2017, which states that UTC – TAI will remain at −37 seconds for the next six months, at least.

2017-01-01 Added a description of the stat function, the list of entry points and the kernel recommendations. Handle converting -2^{127} to a string.

2016-12-21 Original distribution.

12 Future Work

The table of leap seconds will need to be updated every six months, when the IERS releases Bulletin C with a new leap second announcement. These updates can also capture the latest research on the rotation of the Earth in historical times. It would be nice if these updates could be included in the software automatically.

13 Licensing

13.1 You are free to:

13.2 Under the following terms:

Attribution — You must give appropriate credit,[2] provide a link to the license, and indicate if changes were made.[3] You may do so in any reasonable manner, but not in any that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license[4] as the original.

No additional restrictions — You may not apply legal terms or technological measures[5] that legally restrict others from doing anything the license permits.

13.3 Notices:

- You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an exception or limitation.
- No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy or moral rights may limit how you use the material.

[2] If supplied, you must provide the name of the creator and attribution parties, a copyright notice, a license notice, a disclaimer notice, and a link to the material.

[3] You must indicate if you modified the material and retain an indication of previous modifications.

[4] You may also use any of the licenses listed as compatible at the following web site: `https://creativecommons.org/compatiblelicenses`

[5] The license prohibits application of effective technological measures, defined with reference to Article 11 of the WIPO Copyright Treaty.

References

[1] Standard for Information Technology–Portable Operating System Interface (POSIX®) Base Specifications, Issue 7. *IEEE Std 1003.1, 2016 Edition (incorporates IEEE Std 1003.1-2008, IEEE Std 1003.1-2008/Cor 1-2013, and IEEE Std 1003.1-2008/Cor 2-2016)*, pages 1–3957, Sept 2016. doi: 10.1109/IEEESTD.2016.7582338. 2

[2] D. D. McCarthy and A. K. Babcock. The length of day since 1656. *Physics of the Earth and Planetary Interiors*, 44:281–292, November 1986. doi: 10.1016/0031-9201(86)90077-4. URL `http://adsabs.harvard.edu/abs/1986PEPI...44..281M`. Provided by the SAO/NASA Astrophysics Data System. 66

[3] L. V. Morrison and F. R. Stephenson. Historical values of the Earth's clock error ΔT and the calculation of eclipses. *Journal for the History of Astronomy*, 35:327–336, August 2004. URL `http://adsabs.harvard.edu/abs/2004JHA....35..327M`. Provided by the SAO/NASA Astrophysics Data System. 66

[4] L. V. Morrison and F. R. Stephenson. Addendum: Historical values of the Earth's clock error. *Journal for the History of Astronomy*, 36:339, August 2005. URL `http://adsabs.harvard.edu/abs/2005JHA....36..339M`. Provided by the SAO/NASA Astrophysics Data System. 66

[5] John Sauter. Extending Coordinated Universal Time to Dates Before 1972. *Wikimedia*, January 2017. URL `https://commons.wikimedia.org/wiki/File:Extending_Coordinated_Universal_Time_to_Dates_Before_1972.pdf`. 12

[6] F. R. Stephenson. Historical Eclipses and Earth's Rotation: 700 BC–AD 1600. *Astrophysics and Space Science Proceedings*, 23:3, 2011. doi: 10.1007/978-1-4419-8161-5_1. URL `http://adsabs.harvard.edu/abs/2011ASSP..23....3S`. Provided by the SAO/NASA Astrophysics Data System. 66

[7] F. R. Stephenson, J. E. Jones, and L. V. Morrison. The solar eclipse observed by Clavius in A.D. 1567. *Astronomy and Astrophysics*, 322:347–351, June 1997. URL `http://adsabs.harvard.edu/abs/1997A%26A...322..347S`. Provided by the SAO/NASA Astrophysics Data System. 66

[8] F. R. Stephenson, L. V. Morrison, and C. Y. Hohenkerk. Measurement of the Earth's rotation: 720 BC to AD 2015. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 472 (2196), 2016. ISSN 1364-5021. doi: 10.1098/rspa.2016.0404. URL `http://rspa.royalsocietypublishing.org/content/472/2196/20160404`. 66