Avoid Using POSIX `time_t` for Telling Time[*]

John Sauter[†]

2025-09-19

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.

URL: https://www.systemeyescomputerstore.com/leap_seconds/avoid_time_t.pdf

[†]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 midnight Greenwich Mean Time on January 1, 1970. This count of seconds since the Epoch is also defined as an encoding of Coordinated Universal Time into an integer.

    The following subsection is the formal definition of "Seconds since the Epoch" from POSIX:

<div align="center">

* * * * * Beginning of POSIX text * * * * *

</div>

4.19 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 $\geq$ 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 date and time in Coordinated Universal Time, as determined by the International Earth Rotation Service, and the system's 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.

<div align="center">

* * * * * End of POSIX text * * * * *

</div>

2   Problems

There are several 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 POSIX `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:

$$value = (year \times 10000000000) + (month \times 100000000) +$$
$$(day \times 1000000) + (hour \times 10000) + (minute \times 100) + second$$

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 for more than a minute, so application programmers will not be tempted to misuse it for that purpose.

  An alternative coding of time as an integer is based on International Atomic Time. See section 6.

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, CLOCK_BOOTTIME can be used, since it is a count of seconds since an arbitrary epoch. Unlike `time_t`, CLOCK_BOOTTIME does not change its epoch while applications are running, and so it never appears to flow backwards. However, CLOCK_-BOOTTIME 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. For the convenience of the reader I have packaged the demonstration code below into a library called `libtime`. See section 9 for details.

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;
struct timeval current_timeval;
int adjtimex_result;

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

if (adjtimex_result == -1)
  {
    /* Some high security environments disable the adjtimex function,
     * even when, as here, it is just fetching information.  Fall back
     * to using gettimeofday.  This is a poor fallback, since it does
     * not tell us that we are in a leap second, but it is better than
     * failing. */
    gettimeofday (&current_timeval, NULL);
    gmtime_r (&current_timeval.tv_sec, current_tm);
    *nanoseconds = current_timeval.tv_usec * 1e3;
    return (0);
  }

/* Format the information returned by adjtimex 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);
}
```

Notice in the code above the workaround for `adjtimex` not working. I feel that it is intolerable for an environment to conceal the fact that a leap second is in progress, so I recommend that your application refuse to run in such an

environment. You can do that by calling the following subroutine at the start of
your program.

```c
int
time_test_for_disabled_adjtimex ()
{
  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);

  if (adjtimex_result == -1)
    {
      /* Some high security environments disable the adjtimex function,
       * even when, as here, it is just fetching information.
       * The adjtimex function is the only way to determine that
       * there is a leap second in progress, so return 1 if it
       * doesn't work.  */
      return (1);
    }

  /* Return 0 to indicate that adjtimex has not been disabled.  */
  return (0);
}
```

If the subroutine returns a non-zero result, your applicaton should complain
and exit:

```c
static int
test_for_disabled_adjtimex ()
{
  int adjtimex_is_disabled;

  adjtimex_is_disabled = time_test_for_disabled_adjtimex();

  if (adjtimex_is_disabled != 0)
    {
      /* Some high security environments disable the adjtimex function,
       * even when, as here, it is just fetching information.
       * The adjtimex function is the only way to determine that
       * there is a leap second in progress, so complain if it
       * doesn't work.  */
      printf ("The current time will not be correct during a leap second\n"
              "because the Linux adjtimex function is not working.\n");
```

```
        exit (EXIT_FAILURE);
    }

  return (0);
}
```

The above subroutine is not included in the library, though the library does contain `time_test_for_disabled_adjtimex`. Feel free to express yourself as you see fit about the disabling of `adjtimex`. Fedora bugzilla 1 778 298[1] contains my opinion.

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_BOOTTIME, since CLOCK_BOOTTIME 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_year)
{
  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_year);

  /* 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;
```

---

[1]https://bugzilla.redhat.com/show_bug.cgi?id=1778298

```
  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_year.

4.4   clock_settime

Clock_settime is not used by applications. However, see section 12 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.

```
/* Subroutine to compute the difference, in seconds, between two
 * instants of Coordinated Universal time.
 *
 * The parameter variable_length_seconds_before_year controls
 * the interpretation of seconds before the specified year.
 * 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 contemporary 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.
 *
 * If you want to use fixed-length seconds for all dates, set
 * variable_length_seconds_before_year to INT_MIN.  If you want
 * fixed-length seconds starting January 1, 1972, but variable-length
 * seconds before January 1, 1972, set variable_length_seconds_before_year
 * to 1972.  If you want to ignore leap seconds entirely, set
 * variable_length_seconds_before_year to INT_MAX.
 */
long long int
time_diff (struct tm *A_tm, struct tm *B_tm,
           int variable_length_seconds_before_year)
{
  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_year);
    }

  /* 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.
 * A Julian Day starts at noon, so the Julian Day Number that
 * marks the start of a calendar day ends with .5.  We are only
 * concerned with days as a whole, and we would like to work with
 * integers, so we drop the .5 from the Julian Day Number that
 * marks the start of the calendar day.  If you wish to use the
 * number returned by this subroutine as a proper Julian Day
 * Number, append .5 to it.
 */
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 - 32046;
  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_year)
{
  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_year);
  B_dtai = time_DTAI (B_jdn,
                      variable_length_seconds_before_year);

  /* 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_year)
{
  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 a particular year, then the value of DTAI
   * for all dates before that year is the same as the value
   * for January 1 of that year.  If an application wants no
   * variable-length seconds, it should specify INT_MIN.
   * If an application wants all seconds to be variable-length,
   * it should specify INT_MAX.
   */

  switch (variable_length_seconds_before_year) {
    case INT_MIN:
      /* No variable-length seconds. */
      the_day = day_number;
```

```
        break;

    default:
      /* No variable-length seconds before a specified year.  */
      the_day = day_number;
      if (the_day < time_Julian_day_number (
                        variable_length_seconds_before_year,
                        1, 1))
        {
          the_day = time_Julian_day_number (
                        variable_length_seconds_before_year,
                        1, 1);
        }
      break;

    case INT_MAX:
      /* No leap seconds.  */
      return (0);
      break;
}

/* Variable the_day is the date, expressed as a Julian day number,
 * whose DTAI value we need.  If the table includes the date,
 * do a binary search on the table.  If the day is outside the table,
 * extrapolate.
 */
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:

```
{2441499, 11}, /* 1 Jul 1972 */
{2441683, 12}, /* 1 Jan 1973 */
{2442048, 13}, /* 1 Jan 1974 */
{2442413, 14}, /* 1 Jan 1975 */
{2442778, 15}, /* 1 Jan 1976 */
{2443144, 16}, /* 1 Jan 1977 */
{2443509, 17}, /* 1 Jan 1978 */
{2443874, 18}, /* 1 Jan 1979 */
```

```
{2444239, 19}, /* 1 Jan 1980 */
{2444786, 20}, /* 1 Jul 1981 */
{2445151, 21}, /* 1 Jul 1982 */
{2445516, 22}, /* 1 Jul 1983 */
{2446247, 23}, /* 1 Jul 1985 */
{2447161, 24}, /* 1 Jan 1988 */
{2447892, 25}, /* 1 Jan 1990 */
{2448257, 26}, /* 1 Jan 1991 */
{2448804, 27}, /* 1 Jul 1992 */
{2449169, 28}, /* 1 Jul 1993 */
{2449534, 29}, /* 1 Jul 1994 */
{2450083, 30}, /* 1 Jan 1996 */
{2450630, 31}, /* 1 Jul 1997 */
{2451179, 32}, /* 1 Jan 1999 */
{2453736, 33}, /* 1 Jan 2006 */
{2454832, 34}, /* 1 Jan 2009 */
{2456109, 35}, /* 1 Jul 2012 */
{2457204, 36}, /* 1 Jul 2015 */
{2457754, 37}, /* 1 Jan 2017 */
{2462683, 36}, /* 1 Jul 2030 */
{2467250, 37}, /* 1 Jan 2043 */
```

The left number is the integer part of 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 9. The table can be generated automatically[6], 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[2] announcing the next leap second. See file `extraordinary_days.dat`, also embedded in this PDF file, for details.

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. If you want the local time in a foreign location you can also specify the UTC offset.

---

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

```c
/* Convert a tm structure containing Coordinated Universal Time
 * to one containing a foreign local time.  Foreign_UTC_offset is
 * the offset in seconds between UTC and the desired local time.
 */
int
time_UTC_to_foreign_local (struct tm *coordinated_universal_time,
                           int foreign_UTC_offset,
                           struct tm *local_time_tm,
                           int variable_length_seconds_before_year)
{
  struct tm utc_time_tm;
  int gmt_offset, hours_offset, minutes_offset, seconds_offset;
  int prev_minute_length;

  time_copy_tm (coordinated_universal_time, &utc_time_tm);
  gmt_offset = foreign_UTC_offset;

  /* 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_year);
      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 the other fields in the tm structure do not contain garbage.  */
  local_time_tm->tm_wday = 0;
  local_time_tm->tm_yday = 0;
  local_time_tm->tm_isdst = -1;
  local_time_tm->tm_gmtoff = gmt_offset;
  local_time_tm->tm_zone = NULL;

  /* Make sure all the fields are in their valid ranges
   * and update the fields output by mktime.  */
  time_local_normalize (local_time_tm, local_time_tm->tm_sec,
                        variable_length_seconds_before_year);

  return (0);
}

/* 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_year)
{
  struct tm utc_time_tm;
```

```
    struct tm temporary_local_time_tm;
    time_t seconds_since_epoch;
    int gmt_offset;

    /* 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, &temporary_local_time_tm);
    gmt_offset = temporary_local_time_tm.tm_gmtoff;

    /* Now that we have access to the GMT offset, we do the
     * conversion again, this time taking leap seconds into
     * account.  */
    return (time_UTC_to_foreign_local (coordinated_universal_time,
                                       gmt_offset,
                                       local_time_tm,
                                       variable_length_seconds_before_year));
}
```

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_year)
{
  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 other fields in the tm structure do not contain garbage.  */
    utc_time_tm.tm_wday = 0;
    utc_time_tm.tm_yday = 0;
    utc_time_tm.tm_isdst = -1;
    utc_time_tm.tm_gmtoff = 0;
    utc_time_tm.tm_zone = NULL;

    /* Make sure the fields of the tm structure are all in their
     * valid ranges and update the fields output by mktime.  */
    time_UTC_normalize (&utc_time_tm, utc_time_tm.tm_sec,
                        variable_length_seconds_before_year);

    /* 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 12 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 12 for a recommendation for making the file times accurate.

4.13 strftime

Strftime has formatting code %s, which outputs the number of seconds since the Epoch. Don't use this.

4.14 time

Time returns the current time as a `time_t`. Instead, 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;
  struct timeval current_timeval;
  int adjtimex_result;

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

  if (adjtimex_result == -1)
    {
      /* Some high security environments disable the adjtimex function,
       * even when, as here, it is just fetching information.  Fall back
       * to using gettimeofday.  This is a poor fallback, since it does
       * not tell us that we are in a leap second, but it is better than
       * failing. */
      gettimeofday (&current_timeval, NULL);
      gmtime_r (&current_timeval.tv_sec, current_tm);
```

```
      return (0);
    }
  /* 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);
}
```

4.15   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. Note that this code depends on having a C compiler which supports 128-bit integers.

```
/* 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. Note that this code depends on having a C compiler which supports 128-bit integers.

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

```c
 * not counting the trailing NUL character.
 * Value_p is a pointer to the integer to be converted.
 * Result is a pointer to the output 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_p, 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;
  volatile __int128 volatile_value;
  __int128 value;

  size_remaining = result_size;  /* Guard against exceeding the size
                                  * of the result string.  */
  result_pointer = result;
  character_count = 0;  /* Keep track of the number of characters
                         * we write, so we can return it.  */

  /* Don't require the value parameter to be 16-byte aligned.
   * Python represents a 128-bit number as a structure of two
   * 64-bit numbers, and so does not know that such a structure
   * should be 16-byte aligned.  */
  memcpy (&value, value_p, sizeof(value));

  /* If the value is negative, produce a minus sign
   * and then produce digits for 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;
        }
```

```c
  }
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;
        character_count = character_count + 1;
        *result_pointer = 0;
      }
    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.
 * We use a volatile variable to force the compiler to execute
 * the test, because otherwise the compiler might assume
 * that a negative value, after being negated, is necessarily
 * positive.  */
volatile_value = positive_value;
if (volatile_value < 0)
  {
    current_digit = 8;
    /* The compiler will not let us say
    remaining_digits =
      (__int128) 17014118346046923173168730371588410572;
      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;
      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;
    }

  /* 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.

```c
/* 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.

```c
/* 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_year)
{
  /* 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_year))
    {
      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;
                        }
                    }
                }
            }
```

```c
            break;

        case -1:
          /* Round down to the last second of this day.  */
          time_tm->tm_sec =
            time_length_UTC_minute (time_tm,
                                    variable_length_seconds_before_year)
            - 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_year)
{
  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;
```

```c
        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_year);
  if (return_value != 0)
    return (return_value);

  return_value =
    time_UTC_normalize (time_tm, time_tm->tm_sec,
                        variable_length_seconds_before_year);
  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_year)
{
  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;
```

```c
  }

  /* 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_year);
  if (return_value != 0)
    return (return_value);

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

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

```c
time_UTC_add_days (struct tm *time_tm, int addend,
                   int rounding_mode,
                   int variable_length_seconds_before_year)
{
  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_year);
  if (return_value != 0)
    return (return_value);

  return_value =
    time_UTC_normalize (time_tm, time_tm->tm_sec,
                        variable_length_seconds_before_year);
```

```c
  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_year)
{
  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_year);
  if (return_value != 0)
    return (return_value);

  return_value =
    time_UTC_normalize (time_tm, time_tm->tm_sec,
                        variable_length_seconds_before_year);
  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_year)
{
  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;
                }
            }
        }
    }
```

```c
    }

  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_year);
  if (return_value != 0)
    return (return_value);

  return_value =
    time_UTC_normalize (time_tm, time_tm->tm_sec,
                        variable_length_seconds_before_year);
  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_year)
{
  long long int add_nanoseconds;
  long long int nanoseconds;
  int return_value;
```

```c
  /* 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_year);
  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_year)
{
  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_year);
*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);
}

/* 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_year)
{

  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_year);

  /* Do the same for tomorrow.  */
  JDN_tomorrow = JDN_today + 1;
  DTAI_tomorrow = time_DTAI (JDN_tomorrow,
                             variable_length_seconds_before_year);

  /* 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_year)
{
  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_UTC_minute (&prev_minute_tm,
                            variable_length_seconds_before_year);
  return (return_value);
}

/* Return the length of this local minute.  */
int
time_length_local_minute (struct tm *time_tm,
                          int variable_length_seconds_before_year)
{
  /* Modern time zones are mostly a multiple of 60 minutes offset
   * from UTC.  Those that are not are mostly a multiple of 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_year);
    return_val =
      time_length_UTC_minute (&utc_time_tm,
                              variable_length_seconds_before_year);
    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_year)
{
    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)
              {
```

```c
              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_year);
  return (return_value);
}
```

The subroutines above also use time_UTC_normalize.

```c
/* 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_year)
{
  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_year);
        time_tm->tm_min = time_tm->tm_min - 1;
        continue;
      }

  if (seconds >=
      time_length_UTC_minute (time_tm,
                          variable_length_seconds_before_year))
      {
        seconds = seconds -
          time_length_UTC_minute (time_tm,
                          variable_length_seconds_before_year);
        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_year)
{

  /* 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_year))
    {
      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_year)
            - 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_year)
{
  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_year);
  if (return_value != 0)
    return (return_value);

  return_value =
    time_local_normalize (time_tm, time_tm->tm_sec,
                          variable_length_seconds_before_year);
  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_year)
{
  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)
    {
```

```c
      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_year);
  if (return_value != 0)
    return (return_value);

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

```
                                variable_length_seconds_before_year);
    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_year)
{
  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_year);
  if (return_value != 0)
```

```c
    return (return_value);

  return_value =
    time_local_normalize (time_tm, time_tm->tm_sec,
                          variable_length_seconds_before_year);
  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_year)
{
  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_year);
  if (return_value != 0)
    return (return_value);

  return_value =
    time_local_normalize (time_tm, time_tm->tm_sec,
                          variable_length_seconds_before_year);
  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_year)
{
  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_year);
  if (return_value != 0)
    return (return_value);

  return_value =
    time_local_normalize (time_tm, time_tm->tm_sec,
                          variable_length_seconds_before_year);
  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,
```

```c
                            int variable_length_seconds_before_year)
{
  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_year);
  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_year)
{
  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
   * designated 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_year);
  *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_year)
{
  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_year);
        time_tm->tm_min = time_tm->tm_min - 1;
        continue;
      }

    if (seconds >=
        time_length_local_minute (time_tm,
                          variable_length_seconds_before_year))
      {
        seconds = seconds -
          time_length_local_minute (time_tm,
                          variable_length_seconds_before_year);
        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,
```

```c
                    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   International Atomic Time

Applications that are more concerned with measuring time intervals than with the rotation of the Earth would like a data type like the POSIX `time_t` but without leap seconds. A common way to construct such a data type is with reference to International Atomic Time, or TAI.

TAI is a time scale which started on January 1, 1958, at 00:00:00 GMT and has counted fixed-length SI seconds since then. For the 14 years from 1958 to 1972, UTC used seconds which averaged shorter than the SI seconds, so by 1972 the two time scales differed by about 10 seconds. During 1971 the UTC time scale was made to exactly match the TAI time scale, with an offset of 10 seconds.

Since January 1, 1972 at 00:00:00 UTC, both TAI and UTC have counted SI seconds, each in its own way. UTC inserts leap seconds to keep civil time synchronized to the rotation of the Earth, and TAI does not. Each time UTC inserts a leap second, the difference between TAI and UTC increases by one second.

UNIX programmers like to use 1970 as a base date, so they backdated the synchronization of TAI and UTC to January 1, 1970, but kept the 10-second offset. Thus, the new data type, which I will call the TAI integer, is the number of SI seconds since January 1, 1970, plus 10.

The benefit of this data type is that it makes computing the number of seconds between two times after January 1, 1972 easy: just subtract the integers. The tradeoff is that displaying the time in UTC is more difficult. For example, the TAI integer 1 483 228 836 corresponds to December 31, 2016, at 23:59:60. See the example in subsection 10.4 for this conversion.

7   List of Entry Points

For reference, here are the subroutines described above, in alphabetical order, each with a brief description. The attribute "const" means that the function does not examine any values except its arguments, and has no effects except to return a value. The attribute "pure" means the same except that the function can examine (but not change) global variables.

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

/* 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_year)
  __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_year)
  __attribute__ ((const));

/* Compute the Julian Day Number corresponding to a date.
 * Note that this returns just the integer part of the
 * Julian Day Number.  Add 0.5 to get the real Julian Day
 * Number, or subtract 2400000 to get the Modified Julian
 * Day Number (MJD).  */
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_year)
  __attribute__ ((pure));

/* 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_year)
```

```c
  __attribute__ ((pure));

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

/* 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_year)
  __attribute__ ((pure));

/* 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_year)
  __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_year);

/* 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_year);

/* 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_year);

/* 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_year);

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

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

/* 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_year);

/* 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_year);

/* Make sure all of the fields of a tm structure containing
 * local time are within their valid ranges.  Also set tm_wday to
 * the day of the week and tm_yday to the day of the year for the date
 * as specified by the year (tm_year), month (tm_mon) and day of the
 * month (tm_mday).  Whenever a tm structure is changed by these
 * subroutines it is normalized before it is returned.  */
int
time_local_normalize (struct tm *time_tm,
                      long long int seconds,
                      int variable_length_seconds_before_year);

/* 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_year);

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

/* Test the adjtimex function to see if it has been disabled.  */
int
time_test_for_disabled_adjtimex ();

/* 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_year);

/* 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_before_year);

/* 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_year);

/* 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_year);

/* 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_year);

/* Add seconds and nanoseconds to a Coordinated Universal 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_year);

/* Add years to a Coordinated Universal Time.  */
int
time_UTC_add_years (struct tm *time_tm, int addend,
                    int rounding_mode,
                    int variable_length_seconds_before_year);

/* Make sure all of the fields of a tm structure containing a
 * Coordinated Universal Time are within their valid ranges.
 * Also set tm_wday to the day of the week and tm_yday to the
 * day of the year for the date as specified by the year (tm_year),
 * month (tm_mon) and day of the month (tm_mday).
 * Whenever a tm structure is changed by these subroutines
 * it is normalized before it is returned.  */
int
time_UTC_normalize (struct tm *time_tm,
                    long long int seconds,
                    int variable_length_seconds_before_year);

/* Convert a tm structure containing Coordinated Universal Time
 * to one containing a foreigh local time.  Foreign_UTC_offset is
 * the offset in seconds between UTC and the desired local time.
 */
int
time_UTC_to_foreign_local (struct tm *coordinated_universal_time,
                           int foreign_UTC_offset,
                           struct tm *local_time_tm,
                           int variable_length_seconds_before_year);

/* Convert a Coordinated Universal Time to local time.  */
```

```
int
time_UTC_to_local (struct tm *coordinated_universal_time,
                   struct tm *local_time,
                   int variable_length_seconds_before_year);
```

8 Python

The Python module datetime contains a class also called datetime which ignores leap seconds, and therefore has the same problem as POSIX time_t. A program written in the Python programming language must therefore refrain from using datetime in order to handle leap seconds correctly.

You can use the POSIX tm data structure in place of datetime, and use these same C subroutines to manipulate it using ctypes. Write a module to define the interface, as follows:

```python
import ctypes
import ctypes.util

# Translation into ctypes of the POSIX tm structure
# as implemented on GNU/Linux.

class tm (ctypes.Structure):
    _fields_ = [
        ("tm_sec", ctypes.c_int),
        ("tm_min", ctypes.c_int),
        ("tm_hour", ctypes.c_int),
        ("tm_mday", ctypes.c_int),
        ("tm_mon", ctypes.c_int),
        ("tm_year", ctypes.c_int),
        ("tm_wday", ctypes.c_int),
        ("tm_yday", ctypes.c_int),
        ("tm_isdst", ctypes.c_int),
        ("tm_pad1", ctypes.c_int),
        ("tm_gmtoff", ctypes.c_long),
        ("tm_zone", ctypes.POINTER(ctypes.c_char_p))
    ]
    def __str__(self):
        string_p = ctypes.create_string_buffer (256)
        time_tm_to_string (self, string_p, ctypes.sizeof(string_p))
        return (string_p.value.decode("utf-8"))

# A 128-bit integer.
class int128 (ctypes.Structure):
    _fields_ = [
```

```python
            ("low_bits", ctypes.c_ulonglong),
            ("high_bits", ctypes.c_longlong)
    ]
    def __str__(self):
        string_p = ctypes.create_string_buffer (256)
        int128_to_string (self, string_p, ctypes.sizeof (string_p))
        return (string_p.value.decode("utf-8"))


# Note that if you install libtime in /usr/local/lib, which is the
# default location if you install from the tarball, you will need
# to set LD_LIBRARY_PATH to include it.
library_name = ctypes.util.find_library("time")
sub = ctypes.CDLL(library_name)


# Translation into ctypes of time_subroutines.h:

sub.int128_to_string.argtypes = [ctypes.POINTER(int128),
                                 ctypes.c_char_p, ctypes.c_int]
"""Convert a 128-bit integer to a string."""
def int128_to_string (value, result_p, result_size):
    return (sub.int128_to_string (value, result_p, result_size))


sub.time_copy_tm.argtypes = [ctypes.POINTER(tm),
                             ctypes.POINTER(tm)]
"""Copy a tm structure."""
def time_copy_tm (in_tm, out_tm):
    return (sub.time_copy_tm (in_tm, out_tm))


sub.time_current_tm.argtypes = [ctypes.POINTER(tm)]
"""Fetch the current time, without nanoseconds."""
def time_current_tm (tm):
    return (sub.time_current_tm (tm))


sub.time_current_tm_nano.argtypes = [ctypes.POINTER(tm),
                                     ctypes.POINTER(ctypes.c_int)]
"""Fetch the current time, including nanoseconds."""
def time_current_tm_nano (tm):
    result_p = ctypes.create_string_buffer (ctypes.sizeof(ctypes.c_int))
    result_p = ctypes.cast (result_p, ctypes.POINTER(ctypes.c_int))
    return_value = sub.time_current_tm_nano (tm, result_p)
    return (return_value, result_p.contents.value)


sub.time_diff.argtypes = [ctypes.POINTER(tm), ctypes.POINTER(tm),
                          ctypes.c_int]
sub.time_diff.restype = ctypes.c_longlong
"""Compute the difference between two times in seconds."""
```

```python
def time_diff (tm_A, tm_B, variable_length_seconds_before_year):
    return (sub.time_diff (tm_A, tm_B,
                           variable_length_seconds_before_year))

sub.time_DTAI.argtypes = [ctypes.c_int, ctypes.c_int]
"""Fetch the difference between TAI and UTC at the beginning
of the specified Julian Day Number."""
def time_DTAI (Julian_day_number, variable_length_seconds_before_year):
    return (sub.time_DTAI (Julian_day_number,
                           variable_length_seconds_before_year))

sub.time_Julian_day_number.argtypes = [ctypes.c_int, ctypes.c_int,
                                        ctypes.c_int]
"""Compute the Julian Day Number corresponding to a date.
Note that this subroutine returns just the integer part
of the Julian Day Number.  Add 0.5 to get the real Julian
Day Number, or subtract 2400000 to get the Modified Julian
Day Number (MJD)."""
def time_Julian_day_number (year, month, day):
    return (sub.time_Julian_day_number (year, month, day))

sub.time_length_local_minute.argtypes = [ctypes.POINTER(tm),
                                          ctypes.c_int]
"""Compute the length of the current minute of local time."""
def time_length_local_minute (tm, variable_length_seconds_before_year):
    return (sub.time_length_local_minute (
        tm, variable_length_seconds_before_year))

sub.time_length_month.argtypes = [ctypes.POINTER(tm)]
"""Compute the length of the current month."""
def time_length_month (tm):
    return (sub.time_length_month (tm))

sub.time_length_prev_local_minute.argtypes = [ctypes.POINTER(tm),
                                               ctypes.c_int]
"""Compute the length of the previous minute of local time."""
def time_length_prev_local_minute (tm, variable_length_seconds_before_year):
    return (sub.time_length_prev_local_minute (
        tm, variable_length_seconds_before_year))

sub.time_length_prev_month.argtypes = [ctypes.POINTER(tm)]
"""Compute the length of the previous month."""
def time_length_prev_month (tm):
    return (sub.time_length_prev_month (tm))

sub.time_length_prev_UTC_minute.argtypes = [ctypes.POINTER(tm),
```

```
                                                  ctypes.c_int]
"""Compute the length of the previous minute of Coordinated Universal Time."""
def time_length_prev_UTC_minute (tm, variable_length_seconds_before_year):
    return (sub.time_length_prev_UTC_minute (
        tm, variable_length_seconds_before_year))


sub.time_length_UTC_minute.argtypes = [ctypes.POINTER(tm),
                                        ctypes.c_int]
"""Compute the length of the current minute of Coordinated Universal Time."""
def time_length_UTC_minute (tm, variable_length_seconds_before_year):
    return (sub.time_length_UTC_minute (tm,
                                        variable_length_seconds_before_year))


sub.time_local_add_days.argtypes = [ctypes.POINTER(tm), ctypes.c_int,
                                    ctypes.c_int, ctypes.c_int]
"""Add days to a local time."""
def time_local_add_days (tm, addend, rounding_mode,
                         variable_length_seconds_before_year):
    return (sub.time_local_add_days (tm, addend, rounding_mode,
                                     variable_length_seconds_before_year))


sub.time_local_add_hours.argtypes = [ctypes.POINTER(tm), ctypes.c_int,
                                     ctypes.c_int, ctypes.c_int]
"""Add hours to a local time."""
def time_local_add_hours (tm, addend, rounding_mode,
                          variable_length_seconds_before_year):
    return (sub.time_local_add_hours (tm, addend, rounding_mode,
                                      variable_length_seconds_before_year))


sub.time_local_add_minutes.argtypes = [ctypes.POINTER(tm), ctypes.c_int,
                                       ctypes.c_int, ctypes.c_int]
"""Add minutes to a local time."""
def time_local_add_minutes (tm, addend, rounding_mode,
                            variable_length_seconds_before_year):
    return (sub.time_local_add_minutes (
        tm, addend, rounding_mode, variable_length_seconds_before_year))


sub.time_local_add_months.argtypes = [ctypes.POINTER(tm), ctypes.c_int,
                                       ctypes.c_int, ctypes.c_int]
"""Add months to a local time."""
def time_add_months (tm, addend, rounding_mode,
                     variable_length_seconds_before_year):
    return (sub.time_add_months (tm, addend, rounding_mode,
                                 variable_length_seconds_before_year))


sub.time_local_add_seconds.argtypes = [ctypes.POINTER(tm),
```

```python
                                            ctypes.c_longlong,
                                            ctypes.c_int]
"""Add seconds to a local time."""
def time_local_add_seconds (tm, add_seconds,
                            variable_length_seconds_before_year):
    return (sub.time_local_add_seconds (
        tm, add_seconds, variable_length_seconds_before_year))


sub.time_local_add_seconds_ns.argtypes = [
    ctypes.POINTER(tm), ctypes.POINTER(ctypes.c_longlong),
    ctypes.c_longlong, ctypes.c_longlong, ctypes.c_int]
"""Add seconds and nanoseconds to a local time."""
def time_local_add_seconds_ns (tm, add_seconds, add_nanoseconds,
                               variable_length_seconds_before_year):
    result_p = ctypes.create_string_buffer (
        ctypes.sizeof(ctypes.c_longlong))
    result_p = ctypes.cast (result_p, ctypes.POINTER(ctypes.c_longlong))
    return_value = sub.time_local_add_seconds_ns (
        tm, result_p, add_seconds, add_nanoseconds,
        variable_length_seconds_before_year)
    return (return_value, result_p.contents.value)



sub.time_local_add_years.argtypes = [ctypes.POINTER(tm), ctypes.c_int,
                                      ctypes.c_int, ctypes.c_int]
"""Add years to a local time."""
def time_local_add_years (tm, addend, rounding_mode,
                          variable_length_seconds_before_year):
    return (sub.time_local_add_years (tm, addend, rounding_mode,
                                      variable_length_seconds_before_year))


sub.time_local_normalize.argtypes = [ctypes.POINTER(tm), ctypes.c_int,
                                      ctypes.c_int]
"""Make sure all of the fields in a tm structure containing local time
are within their valid ranges.  Also, set tm_wday to
the day of the week and tm_yday to the day of the year for the date
as specified by the year (tm_year), month (tm_mon) and
day of the month (tm_mday).  Whenever a tm structure is changed by these
subroutines it is normalized before it is returned."""
def time_local_normalize (tm, seconds, variable_length_seconds_before_year):
    return (sub.time_local_normalize (tm, seconds,
                                      variable_length_seconds_before_year))


sub.time_local_to_UTC.argtypes = [ctypes.POINTER(tm), ctypes.POINTER(tm),
                                   ctypes.c_int]
"""Convert local time to Coordinated Universal Time."""
```

```python
def time_local_to_UTC (local_time, coordinated_universal_time,
                        variable_length_seconds_before_year):
    return (sub.time_local_to_UTC (local_time, coordinated_universal_time,
                                    variable_length_seconds_before_year))


sub.time_sleep_until.argtypes = [ctypes.POINTER(tm), ctypes.c_int, ctypes.c_int]
"""Sleep until a specified Coordinated Universal Time."""
def time_sleep_until (tm, nanoseconds, variable_length_seconds_before_year):
    return (sub.time_sleep_until (tm, nanoseconds,
                                    variable_length_seconds_before_year))


sub.time_test_for_disabled_adjtimex.argtypes = list();
"""Test the adjtimex function to see if it has been disabled."""
def time_test_for_disabled_adjtimex ():
    return (sub.time_test_for_disabled_adjtimex ())


sub.time_tm_nano_to_integer.argtypes = [ctypes.POINTER(tm), ctypes.c_int,
                                            ctypes.POINTER(int128)]
"""Convert time and nanoseconds to a 128-bit integer."""
def time_tm_nano_to_integer (tm, input_nanoseconds):
    result_p = ctypes.create_string_buffer (ctypes.sizeof(int128))
    result_p = ctypes.cast (result_p, ctypes.POINTER(int128))
    return_value = sub.time_tm_nano_to_integer (tm, input_nanoseconds, result_p)
    return (return_value, result_p.contents)


sub.time_tm_nano_to_string.argtypes = [ctypes.POINTER(tm), ctypes.c_int,
                                            ctypes.c_char_p, ctypes.c_int]
"""Convert the time and nanoseconds to a string."""
def time_tm_nano_to_string (tm, input_nanoseconds, current_time_string,
                            current_time_string_length):
    return (sub.time_tm_nano_to_string (tm, input_nanoseconds,
                                            current_time_string,
                                            current_time_string_length))


sub.time_tm_to_integer.argtypes = [ctypes.POINTER(tm),
                                        ctypes.POINTER(ctypes.c_longlong)]
"""Convert the time to a long long integer."""
def time_tm_to_integer (tm):
    result_p = ctypes.create_string_buffer (
        ctypes.sizeof(ctypes.c_longlong))
    result_p = ctypes.cast (result_p, ctypes.POINTER(ctypes.c_longlong))
    return_value = sub.time_tm_to_integer (tm, result_p)
    return (return_value, result_p.contents.value)


sub.time_tm_to_string.argtypes = [ctypes.POINTER(tm), ctypes.c_char_p,
                                        ctypes.c_int]
```

```python
"""Convert the time to a string."""
def time_tm_to_string (input_tm, current_time_string,
                       current_time_string_length):
    return (sub.time_tm_to_string (input_tm, current_time_string,
                                   current_time_string_length))


sub.time_UTC_add_days.argtypes = [ctypes.POINTER(tm), ctypes.c_int,
                                  ctypes.c_int, ctypes.c_int]
"""Add days to a Coordinated Universal Time."""
def time_UTC_add_days (tm, addend, rounding_mode,
                       variable_length_seconds_before_year):
    return (sub.time_UTC_add_days (tm, addend, rounding_mode,
                                   variable_length_seconds_before_year))


sub.time_UTC_add_hours.argtypes = [ctypes.POINTER(tm), ctypes.c_int,
                                   ctypes.c_int, ctypes.c_int]
"""Add hours to a Coordinated Universal Time."""
def time_UTC_add_hours (tm, addend, rounding_mode,
                        variable_length_seconds_before_year):
    return (sub.time_UTC_add_hours (tm, addend, rounding_mode,
                                    variable_length_seconds_before_year))


sub.time_UTC_add_minutes.argtypes = [ctypes.POINTER(tm), ctypes.c_int,
                                     ctypes.c_int, ctypes.c_int]
"""Add minutes to a Coordinated Universal Time."""
def time_UTC_add_minutes (tm, addend, rounding_mode,
                          variable_length_seconds_before_year):
    return (sub.time_UTC_add_minutes (
        tm, addend, rounding_mode, variable_length_seconds_before_year))


sub.time_UTC_add_months.argtypes = [ctypes.POINTER(tm), ctypes.c_int,
                                    ctypes.c_int, ctypes.c_int]
"""Add months to a Coordinated Universal Time."""
def time_UTC_add_months (tm, addend, rounding_mode,
                         variable_length_seconds_before_year):
    return (sub.time_UTC_add_months (tm, addend, rounding_mode,
                                     variable_length_seconds_before_year))


sub.time_UTC_add_seconds.argtypes = [ctypes.POINTER(tm), ctypes.c_longlong,
                                     ctypes.c_int]
"""Add seconds to a Coordinated Universal Time."""
def time_UTC_add_seconds (tm, add_seconds, variable_length_seconds_before_year):
    return (sub.time_UTC_add_seconds (tm, add_seconds,
                                      variable_length_seconds_before_year))


sub.time_UTC_add_seconds_ns.argtypes = [ctypes.POINTER(tm),
```

```
                                        ctypes.POINTER(ctypes.c_longlong),
                                        ctypes.c_longlong, ctypes.c_longlong,
                                        ctypes.c_int]
"""Add seconds and nanoseconds to a Coordinated Universal Time."""
def time_UTC_add_seconds_ns (tm, add_seconds, add_nanoseconds,
                             variable_length_seconds_before_year):
    result_p = ctypes.create_string_buffer (ctypes.sizeof(ctypes.c_longlong))
    result_p = ctypes.cast (result_p, ctypes.POINTER(ctypes.c_longlong))
    return_value = sub.time_UTC_add_seconds_ns (
        tm, result_p, add_seconds, add_nanoseconds,
        variable_length_seconds_before_year)
    return (return_value, result_p.contents.value)


sub.time_UTC_add_years.argtypes = [ctypes.POINTER(tm), ctypes.c_int,
                                   ctypes.c_int, ctypes.c_int]
"""Add years to a Coordinated Universal Time."""
def time_UTC_add_years (tm, addend, rounding_mode,
                        variable_length_seconds_before_year):
    return (sub.time_UTC_add_years (tm, addend, rounding_mode,
                                    variable_length_seconds_before_year))


sub.time_UTC_normalize.argtypes = [ctypes.POINTER(tm), ctypes.c_longlong,
                                   ctypes.c_int]
"""Make sure all of the fields of a tm structure containing a
Coordinated Universal Time are within their valid ranges.
Also set tm_wday to the day of the week and tm_yday to the
day of the year for the date as specified by the year (tm_year),
month (tm_mon) and day of the month (tm_mday).
Whenever a tm structure is changed by these subroutines
it is normalized before it is returned."""
def time_UTC_normalize (tm, seconds,
                        variable_length_seconds_before_year):
    return (sub.time_UTC_normalize (tm, seconds,
                                    variable_length_seconds_before_year))


sub.time_UTC_to_foreign_local.argtypes = [
    ctypes.POINTER(tm), ctypes.c_int, ctypes.POINTER(tm), ctypes.c_int]

"""Convert a tm structure containing Coordinated Universal Time
to one containing a foreigh local time.  Foreign_UTC_offset is
the offset in seconds between UTC and the desired local time."""
def time_UTC_to_foreign_local (coordinated_universal_time,
                               foreign_UTC_offset,
                               local_time,
                               variable_length_seconds_before_year):
    return (sub.time_UTC_to_foreign_local (
```

```
        coordinated_universal_time, foreign_UTC_offset, local_time,
        variable_length_seconds_before_year))


sub.time_UTC_to_local.argtypes = [ctypes.POINTER(tm), ctypes.POINTER(tm),
                                  ctypes.c_int]


"""Convert a Coordinated Universal Time to local time."""
def time_UTC_to_local (coordinated_universal_time, local_time,
                       variable_length_seconds_before_year):
    return (sub.time_UTC_to_local (coordinated_universal_time,
                                   local_time,
                                   variable_length_seconds_before_year))


# Int_min and int_max are constants used in
# variable_length_seconds_before_year.
int_min = (-ctypes.c_uint(-1).value) // 2
int_max = (ctypes.c_uint(-1).value) // 2
```

This file is provided as time_subroutines.py. Import ctypes and the module, and you can then call the subroutines from Python. Here is a Python script to demonstrate this:

```
# Demonstrate the time_subroutines interface.
import ctypes
import time_subroutines


# Make sure the adjtimex function has not been disabled.
if (time_subroutines.time_test_for_disabled_adjtimex() != 0):
    print ('The current time will not be correct during a leap second')
    print ('because the Linux adjtimex function is not working.')
    raise SystemExit


# Define two variables of type tm.
tm_1 = time_subroutines.tm()
tm_2 = time_subroutines.tm()


# Fetch the current time.
time_subroutines.time_current_tm(tm_1)


# Express the current time as a string.
string_buffer = ctypes.create_string_buffer(256)
time_subroutines.time_tm_to_string(tm_1, string_buffer,
                                   ctypes.sizeof(string_buffer))
the_time = string_buffer.value.decode("utf-8")


# Print the current time.
print (the_time)
```

```python
# Convert the current time to the local time zone.
time_subroutines.time_UTC_to_local (tm_1, tm_2, time_subroutines.int_min)

# Express it as a string using the str method, and print it along with
# the day of the week and the day of the year.
weekday_names = ('Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday',
                 'Friday', 'Saturday')
print (str(tm_2) + ", day of week = " + weekday_names[tm_2.tm_wday] +
       ", day of year = " + str(tm_2.tm_yday) + ".")

# Print the date and time as an integer.
return_value, now_int = time_subroutines.time_tm_to_integer (tm_2)
print ("Corresponding integer = " + str(now_int) + ".")

# Create a time value of December 31, 2016, at 23:59:59 UTC.
tm_3 = time_subroutines.tm (tm_year = 2016 - 1900, tm_mon = 12 - 1,
                            tm_mday = 31, tm_hour = 23, tm_min = 59,
                            tm_sec = 59, tm_isdst = 0, tm_gmtoff = 0,
                            tm_zone = None)

# Calculate the day of the week and the day of the month.
time_subroutines.time_UTC_normalize (tm_3, tm_3.tm_sec,
                                     time_subroutines.int_min)

# Print it.
print (str(tm_3) + ", day of week = " + weekday_names[tm_3.tm_wday] +
       ", day of year = " + str(tm_3.tm_yday) + ".")
return_value, now_int = time_subroutines.time_tm_to_integer (tm_3)
print ("Corresponding integer = " + str(now_int) + ".")

# Add one second.
time_subroutines.time_UTC_add_seconds (tm_3, 1, time_subroutines.int_min)

# And print.  The result should be 2016-12-31T23:59:60.
print (str(tm_3) + ", day of week = " + weekday_names[tm_3.tm_wday] +
       ", day of year = " + str(tm_3.tm_yday) + ".")
return_value, now_int = time_subroutines.time_tm_to_integer (tm_3)
print ("Corresponding integer = " + str(now_int) + ".")

# One more second should get us to the next year.
time_subroutines.time_UTC_add_seconds (tm_3, 1, time_subroutines.int_min)
print (str(tm_3) + ", day of week = " + weekday_names[tm_3.tm_wday] +
       ", day of year = " + str(tm_3.tm_yday) + ".")
return_value, now_int = time_subroutines.time_tm_to_integer (tm_3)
print ("Corresponding integer = " + str(now_int) + ".")
```

Use the Julian Day Number instead of the proleptic Gregorian ordinal from the datetime module, since it covers a wider span of time. If you wish to avoid floating point in your dates, you can use the integer part of the Julian Day Number, with the understanding that 0.5 must be added to get a proper Julian Day Number. The `tm` structure is always associated with a time zone.

There is a Python example in subsection 10.6. It illustrates conversion from POSIX `tm` to Python `time.struct_time`, which is able to represent leap seconds.

9   Embedded files

To save you the trouble of typing in these subroutines, they are embedded in this PDF file. Also included are some sample and test programs, and the configuration scripts needed to compile the programs and construct this PDF file. Building the PDF file requires LaTeX and Python.

9.1   PDF file

If you have only the PDF file, you can build the time library and include files as follows:

1. Install package proleptic_utc_with_leap_seconds to get file `extraordinary_-days.dat` installed. You can find a tarball at this URL: https://github.com/Show-Control/proleptic_utc_with_leap_seconds. Like this library it can also be built from its PDF file, which is at https://www.systemeyescomputer-store.com/leap_seconds/proleptic_UTC.pdf.

2. Create an empty directory.

3. Copy the PDF file to it.

4. Create within that directory subdirectories `examples`, `m4`, `src` and `tests`.

5. Extract the embedded files from the PDF file using pdfdetach -saveall.

6. Type `bash fix_files.sh` to correct the permissions of the extracted files.

7. Build the library the usual way with `./configure` and `make`.

8. Once the library is built you can install it by typing `sudo make install`.

9.2   TAR file

If you have the compressed tar file, extract its contents to an empty directory, then type `./configure`, and `make`. Type `sudo make install` to install the library. If you want the PDF you can get it by typing `make pdf`.

9.3 COPR

If you are running the Fedora distribution of GNU/Linux, the code described in this document is also available in a repository through the Fedora Copr build service. To enable the repository, type `sudo dnf copr enable johnsauter/libtime`. You need to have dnf-plugins-core installed. Once the repository is enabled you can type `sudo dnf install libtime-devel` to install the library, include files and concise documentation. To get this PDF file, install `libtime-doc`. Your users can install `libtime` to get just the library. If you are on a platform whose C compiler does not support 128-bit integers, `int128_to_string` and `time_tm_nano_to_integer` are not included in libtime.

9.4 GitHub

These files are also available on github, at the following URL:

https://github.com/ShowControl/libtime

That directory has all of the code displayed in this document, the LaTeX source for this document, a tarball which holds the sources plus the build and install procedure, and the source RPM.

10 Examples

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

10.1 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, at 9 AM. 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.

10.2 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, INT_MIN);
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, INT_MIN);
    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, INT_MIN);
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, INT_MIN);
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, INT_MIN);
time_local_to_UTC (&extra_time_tm, &int_time_tm, INT_MIN);
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, INT_MIN);
if (diff < 0)
  printf ("Next backup is full.\n");
else
  printf ("Next backup is incremental.\n");
```

10.3   Rocket over Central Park

You are asked to explode a fireworks rocket over Central Park in New York City 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, the extra day at the end of February, 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, INT_MIN);

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, INT_MIN);
    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, INT_MIN);
    time_UTC_add_seconds (&time_tm, -2, INT_MIN);
    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.
```

10.4 Using International Atomic Time

You are writing an application that must compute time intervals quickly, but can afford to spend more time when displaying a time stamp to a person. You decide to use a time scale based on International Atomic Time as described in section 6. You are using the integer representation of TAI to time stamp your logs, and you must now convert those time stamps to readable text.

Construct a base date of January 1, 1970 at 00:00:00. Add the time stamp minus 10 to it, specifying that seconds before 1972 are variable length. Convert

the result to text. The code looks like this:

```
/* Example 4: using TAI.  Time_stamp_value is the TAI integer
 * that is to be converted to a readable format.  */
void
example_4 (long long int time_stamp_value)
{
  struct tm time1_tm;
  struct tm time2_tm;
  char buffer1 [64];

  /* Construct the base date of 1970-01-01T00:00:00Z.  */
  time_current_tm (&time1_tm);
  time1_tm.tm_year = 1970 - 1900;
  time1_tm.tm_mon = 1 - 1;
  time1_tm.tm_mday = 1;
  time1_tm.tm_hour = 0;
  time1_tm.tm_min = 0;
  time1_tm.tm_sec = 0;

  /* Add the time stamp, which is the integer representation of TAI.  */
  time_copy_tm (&time1_tm, &time2_tm);
  time_UTC_add_seconds (&time2_tm, time_stamp_value - 10, 1972);
  time_tm_to_string (&time2_tm, &buffer1 [0], sizeof (buffer1));

  /* Print the result.  */
  printf ("%lld TAI is %s.\n", time_stamp_value, buffer1);
  return;
}
```

If you find the use of variable-length seconds before 1972 troublesome, see the next example.

10.5 Displaying PTP time stamps

IEEE 1588, known as Precision Time Protocol or PTP, counts fixed-length seconds since an epoch. The SMPTE ST-2059-2 profile specifies that the epoch is 63 072 010 seconds before January 1, 1972 at 00:00:00 UTC. That definition avoids the complexity of variable length seconds before 1972, and is effectively the same as the TAI integer in the previous example for dates after January 1, 1972, since there are 63 072 000 variable-length seconds between January 1, 1970 and January 1, 1972.

PTP always measures fixed-length seconds, so how should we display dates between January 1, 1970 and January 1, 1972? I suggest that such times be displayed by pretending that fixed-length seconds were being used during that time. Thus, converting a PTP time to text for display will show a time that

does not correspond to contemporary civil time between January 1, 1970 and January 1, 1972.

The code is similar to the previous example, but using fixed-length seconds before 1972. The display is intended to be more readable than the ISO 8601 format used in the previous example. PTP times 15 638 408 and 47 174 409 correspond to the leap seconds that are needed to keep a count of fixed-length seconds synchronized with the rotation of the Earth.

```c
/* Example 5: using PTP (IEEE 1588) with the SMPTE ST-2059-2 profile.  */
void
example_5 (long long int time_stamp_value)
{
  struct tm time1_tm;
  struct tm time2_tm;
  struct tm time3_tm;
  char buffer1 [128];

  /* Construct the base date of 1972-01-01T00:00:00Z.  */
  time_current_tm (&time1_tm);
  time1_tm.tm_year = 1972 - 1900;
  time1_tm.tm_mon = 1 - 1;
  time1_tm.tm_mday = 1;
  time1_tm.tm_hour = 0;
  time1_tm.tm_min = 0;
  time1_tm.tm_sec = 0;

  /* Add the time stamp, which is an integer.  Note that the epoch
   * for SMPTE ST-2059-2 is 63,072,010 seconds before the base date.
   * PTP always uses fixed-length (SI) seconds.  */
  time_copy_tm (&time1_tm, &time2_tm);
  time_UTC_add_seconds (&time2_tm, time_stamp_value - 63072010, INT_MIN);

  /* Convert to local time and use strftime to make a very readable
   * display.  */
  time_UTC_to_local (&time2_tm, &time3_tm, INT_MIN);
  strftime (&buffer1 [0], sizeof (buffer1),
            "%A, %B %d, %Y, %I:%M:%S %p %Z", &time3_tm);

  /* Print the result.  */
  printf ("PTP %lld displays as %s.\n", time_stamp_value, buffer1);
  return;
}
```

10.6 Displaying PTP time stamps using Python

We wish to display PTP time stamps as in the previous example, but we prefer
to code in Python.

```python
# Example 6: Display PTP time stamps using Python.

import ctypes
import time
import time_subroutines

def example_6 (time_stamp_value):

    # Construct the base date of January 1, 1972, at midnight.
    time1_tm = time_subroutines.tm (
        tm_year = 1972 - 1900, tm_mon = 1 - 1, tm_mday = 1,
        tm_hour = 0, tm_min = 0, tm_sec = 0, tm_isdst = 0,
        tm_gmtoff = 0, tm_zone = None)

    # Add the time stamp, which is an integer.  Note that the epoch
    # for SMPTE ST-2059-2 is 63,072,010 seconds before the base date.
    # PTP always uses fixed-length (SI) seconds.
    time2_tm = time_subroutines.tm()
    time_subroutines.time_copy_tm (time1_tm, time2_tm)
    time_subroutines.time_UTC_add_seconds (
        time2_tm, time_stamp_value - 63072010,
        time_subroutines.int_min)

    # Convert to local time.
    time3_tm = time_subroutines.tm()
    time_subroutines.time_UTC_to_local (
        time2_tm, time3_tm, time_subroutines.int_min)

    # Convert the POSIX tm structure to the Python struct_time structure.
    # Add 1900 to tm_year and 1 to tm_mon.  For tm_wday, struct_time has
    # 0 meaning Monday whereas POSIX has 0 meaning Sunday, thus add 6
    # modulo 7.  For tm_yday, struct_time is 1-366 whereas POSIX is 0-365,
    # so add 1.
    time4 = time.struct_time(
        (time3_tm.tm_year + 1900, time3_tm.tm_mon + 1,
         time3_tm.tm_mday, time3_tm.tm_hour,
         time3_tm.tm_min, time3_tm.tm_sec,
         (time3_tm.tm_wday + 6) % 7,
         time3_tm.tm_yday + 1, time3_tm.tm_isdst))

    # Convert to a string using strftime and print.
    buffer1 = time.strftime ("%A, %B %d, %Y, %I:%M:%S %p %Z", time4)
```

```python
    print ("PTP " + str(time_stamp_value) + " displays as " +
           buffer1 + ".")
    return
```

10.7 Displaying Local Time in Another Time Zone

Howard E. Hinnant has an excellent package of time subroutines for C++ at this URL: https://howardhinnant.github.io/date/tz.html. One of his examples is displaying the local time of arrival of a flight from New York City to Tehran, leaving on December 31, 1978 at 12:01 PM local time and flying for 14 hours and 44 minutes. Local time in New York is 5 hours behind UTC and in Tehran is 3 hours 30 minutes ahead of UTC, so you might expect the plane to arrive on January 1, 1979, at 11:15, but in fact the local time of arrival is 11:14:59 because of the leap second while the plane was in flight.

   Here is Python code to compute the Tehran local time of arrival, assuming it is run at the departure location.

```python
# Example 7: Compute the arrival time of a flight to Tehran.

import ctypes
import time
import time_subroutines

def example_7 (flight_time):

    # Construct the departure date of December 31, 1978,
    # at 12:01 local time.
    time1_tm = time_subroutines.tm()
    time_subroutines.time_current_tm(time1_tm)
    time1_tm.tm_year = 1978 - 1900
    time1_tm.tm_mon = 12 - 1
    time1_tm.tm_mday = 31
    time1_tm.tm_hour = 12
    time1_tm.tm_min = 1
    time1_tm.tm_sec = 0
    time5 = time.struct_time (
        (time1_tm.tm_year + 1900, time1_tm.tm_mon + 1,
         time1_tm.tm_mday, time1_tm.tm_hour,
         time1_tm.tm_min, time1_tm.tm_sec,
         (time1_tm.tm_wday + 6) % 7,
         time1_tm.tm_yday + 1, time1_tm.tm_isdst))
    buffer1 = time.strftime (
        "%A, %B %d, %Y, %I:%M:%S %p %Z", time5)
    print ("The flight leaves here on " + buffer1 + ".")

    # Convert to UTC.
```

```python
    time2_tm = time_subroutines.tm()
    time_subroutines.time_local_to_UTC (
        time1_tm, time2_tm, time_subroutines.int_min)
    # Add the flight time.
    time3_tm = time_subroutines.tm()
    time_subroutines.time_copy_tm (time2_tm, time3_tm)
    time_subroutines.time_UTC_add_seconds (
        time3_tm, flight_time, time_subroutines.int_min)

    # Convert to local time in Tehran.
    # Tehran's time zone is +3:30.
    time4_tm = time_subroutines.tm()
    time_subroutines.time_UTC_to_foreign_local (
        time3_tm, ((3 * 3600) + (30 * 60)),
        time4_tm, time_subroutines.int_min)

    time5 = time.struct_time(
        (time4_tm.tm_year + 1900, time4_tm.tm_mon + 1,
         time4_tm.tm_mday, time4_tm.tm_hour,
         time4_tm.tm_min, time4_tm.tm_sec,
         (time4_tm.tm_wday + 6) % 7,
         time4_tm.tm_yday + 1, time4_tm.tm_isdst))

    # Convert to a string using strftime and print.
    buffer1 = time.strftime (
        "%A, %B %d, %Y, %I:%M:%S %p IRST", time5)
    print ("The flight arrives in Tehran on " +
            buffer1 + ".")
    return

# The flight time is quoted as 14 hours and 44 minutes,
# but that assumes that all minutes are 60 seconds in length.
# In fact, the flight time is the same no matter what the
# clocks say, so we express the flight time as a number of
# seconds.
example_7 ((14 * 3600) + (44 * 60))
```

Note that this code must have the UTC offset and the abbreviation of Tehran's time zone as constants, rather than extracting them from the time zone data base, as Howard's example does.

Another good resource for time zone information is the International Components for Unicode, at http://site.icu-project.org. See the Date/Time Services under ICU User Guide.

11 Limitations

11.1 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 increasingly 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 1825 the measurements become uncertain. Efforts have been made to improve our knowledge of the rotation rate of the Earth in historical times[3][4][8][7][2][9][5]. This software carries a table in file `extraordinary_days.dat` that reaches back to the year -2000 and forward to the year 2500 based on the latest research on the rotation of the Earth. 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 1825 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 might 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.

11.2 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.

12 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 10^9 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.

13 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.

14 Licensing

As noted on the first page, this paper is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License. The code presented here, and embedded in the PDF file, is licensed under the GPL, version 3 or later.

The full text of the Creative Commons Attribution-ShareAlike 4.0 International license is at this web site: https://creativecommons.org/licenses/by-sa/4.0/legalcode, and is embedded in this PDF file. What follows is a human-readable summary of it.

14.1 You are free to:

Share — copy and redistribute the material in any medium or format, and

Adapt — remix, transform, and build upon the material

for any purpose, even commercially. The licensor cannot revoke these freedoms as long as you follow the license terms.

14.2   Under the following terms:

Attribution — You must give appropriate credit,[3] provide a link to the license, and indicate if changes were made.[4] 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[5] as the original.

No additional restrictions — You may not apply legal terms or technological measures[6] that legally restrict others from doing anything the license permits.

14.3   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.

15   Revision History

2025-09-19 5.192.2 Adjust future leap seconds starting in 2161.

2025-08-22 5.191.2 Adjust future leap seconds starting in 2099.

2025-08-15 5.190.2 Adjust future leap seconds starting in 2029. The next leap second is predicted to be a negative leap second on June 30, 2030..

2025-07-25 5.189.2 Adjust future leap seconds starting in 2047.

2025-07-18 5.188.2 Adjust future leap seconds starting in 2051.

2025-07-11 5.187.2 Adjust future leap seconds starting in 2224. Update to IERS Bulletin C 70, issued July 7, 2025, which states that $UTC - TAI$ will remain at -37 seconds for the next six months, at least.

2025-07-04 5.186.2 Adjust future leap seconds starting in 2150.

---

[3]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.

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

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

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

2025-06-13 5.185.2 Adjust future leap seconds starting in 2080.

2025-06-06 5.184.2 Adjust future leap seconds starting in 2194.

2025-05-30 5.183.2 Adjust future leap seconds starting in 2029. The next leap second is predicted to be a negative leap second on December 31, 2029.

2025-05-23 5.182.2 Adjust future leap seconds starting in 2208.

2025-04-04 5.181.2 Adjust future leap seconds starting in 2063.

2025-02-21 5.180.2 Adjust future leap seconds starting in 2063.

2025-01-10 5.179.2 Adjust future leap seconds starting in 2274. Update to IERS Bulletin C 69, issued January 6, 2025, which states that $UTC - TAI$ will remain at -37 seconds for the next six months, at least.

2025-01-03 5.178.2 Adjust future leap seconds starting in 2223.

2024-12-27 5.177.2 Adjust future leap seconds starting in 2141.

2024-12-20 5.176.2 Adjust future leap seconds starting in 2097.

2024-11-01 5.175.2 Adjust future leap seconds starting in 2095.

2024-09-27 5.174.2 Adjust future leap seconds starting in 2142.

2024-09-06 5.173.2 Adjust future leap seconds starting in 2249. Update the POSIX reference to 2024.

2024-08-09 5.172.2 Adjust future leap seconds starting in 2249.

2024-08-02 5.171.2 Adjust future leap seconds starting in 2080.

2024-07-26 5.170.2 Adjust future leap seconds starting in 2275.

2024-07-19 5.169.2 Adjust future leap seconds starting in 2224.

2024-07-05 5.168.2 Adjust future leap seconds starting in 2051.

2024-06-28 5.167.2 Adjust future leap seconds starting in 2332.

2024-06-07 5.166.2 Adjust future leap seconds starting in 2372.

2024-05-31 5.165.2 Adjust future leap seconds starting in 2332.

2024-05-25 5.164.2 Adjust future leap seconds starting in 2218.

2024-05-17 5.163.2 Adjust future leap seconds starting in 2191.

2024-05-10 5.162.2 Adjust future leap seconds starting in 2047.

2024-04-26 5.161.2 Adjust future leap seconds starting in 2067.

2024-04-12 5.160.2 Adjust future leap seconds starting in 2029. The next leap second is predicted to be a negative leap second on June 30, 2029.

2024-04-05 5.159.2 Adjust future leap seconds starting in 2095.

2024-03-22 5.158.2 Adjust future leap seconds starting in 2278.

2024-03-15 5.157.2 Adjust future leap seconds starting in 2081.

2024-03-01 5.156.2 Adjust future leap seconds starting in 2235.

2024-02-23 5.155.2 Adjust future leap seconds starting in 2029. The next leap second is predicted to be a negative leap second on December 31, 2029.

2024-02-16 5.154.2 Adjust three future leap seconds starting in 2029. The next leap second is predicted to be a negative leap second on June 30, 2030.

2024-02-09 5.153.2 Adjust future leap seconds starting in 2405.

2024-02-02 5.152.2 Adjust future leap seconds starting in 2213.

2024-01-26 5.151.2 Adjust future leap seconds starting in 2211.

2024-01-19 5.150.2 Adjust future leap seconds starting in 2185.

2024-01-12 5.149.2 Adjust future leap seconds starting in 2308.

2024-01-05 5.148.2 Adjust future leap seconds starting in 2308.

2023-12-29 5.147.2 Adjust future leap seconds starting in 2189.

2023-12-22 5.146.2 Adjust future leap seconds starting in 2140.

2023-12-15 5.145.2 Adjust future leap seconds starting in 2170.

2023-12-08 5.144.2 Adjust future leap seconds starting in 2199.

2023-12-01 5.143.2 Adjust future leap seconds starting in 2140.

2023-11-25 5.142.2 Adjust future leap seconds starting in 2189.

2023-11-17 5.141.2 Adjust future leap seconds starting in 2056.

2023-11-03 5.140.2 Adjust future leap seconds starting in 2081.

2023-10-29 5.139.2 Adjust future leap seconds starting in 2060.

2023-10-28 5.138.2 Adjust future leap seconds starting in 2439.

2023-10-20 5.137.2 Adjust future leap seconds starting in 2029. The next leap second is predicted to be a negative leap second on December 31, 2029.

2023-10-13 5.136.2 Adjust future leap seconds starting in 2400.

2023-10-07 5.135.2 Adjust future leap seconds starting in 2029. The next leap second is predicted to be a negative leap second on June 30, 2029.

2023-09-29 5.134.2 Adjust future leap seconds starting in 2029. The next leap second is predicted to be a positive leap second on on December 31, 2047. Remove support of RHEL, which only worked for RHEL 8.

2023-09-22 5.133.2 Adjust future leap seconds starting in 2029. The next leap second is predicted to be a negative leap second on June 30, 2029.

2023-09-17 Remove redundant code in int128_to_string found by Facebook / infer.

2023-09-15 5.132.2 Adjust future leap seconds starting in 2034. The next leap second is predicted to be a negative leap second on December 31, 2034.

2023-09-08 5.131.2 Adjust future leap seconds starting in 2419.

2023-09-01 5.130.2 Adjust future leap seconds starting in 2419.

2023-08-18 5.129.2 Adjust future leap seconds starting in 2132.

2023-08-11 5.128.2 Adjust future leap seconds starting in 2407.

2023-08-04 5.127.2 Adjust future leap seconds starting in 2034. The next leap second is predicted to be a negative leap second on December 31, 2034.

2023-07-28 5.126.2 Adjust future leap seconds starting in 2271.

2023-07-21 5.125.2 Adjust future leap seconds starting in 2033. The next leap second is predicted to be a negative leap second on June 30, 2034.

2023-07-14 5.124.2 Adjust future leap seconds starting in 2167.

2023-07-07 5.123.2 Adjust future leap seconds starting in 2082.

2023-06-30 5.122.2 Adjust future leap seconds starting in 2328.

2023-06-23 5.121.2 Adjust future leap seconds starting in 2033. The next leap second is predicted to be a negative leap second on December 31, 2033.

2023-06-16 5.120.2 Adjust future leap seconds starting in 2067.

2023-06-09 5.119.2 Adjust future leap seconds starting in 2172.

2023-06-02 5.118.2 Adjust future leap seconds starting in 2032. The next leap second is predicted to be a negative leap second on June 30, 2033.

2023-05-26 5.117.2 Adjust future leap seconds starting in 2032. The next leap second is predicted to be a negative leap second on December 31, 2032.

2023-05-19 5.116.2 Adjust future leap seconds starting in 2184.

2023-05-13 5.115.2 Copy editing and fix overflow of Python code beyond the margin.

2023-05-12 5.115.2 Adjust future leap seconds starting in 2031. The next leap second is predicted to be a negative leap second on June 30, 2032.

2023-05-05 5.114.2 Adjust future leap seconds starting in 2031. The next leap second is predicted to be a negative leap second on December 31, 2031.

2023-04-28 5.113.2 Adjust future leap seconds starting in 2119.

2023-04-21 5.112.2 Adjust future leap seconds starting in 2119.

2023-04-14 5.111.2 Adjust future leap seconds starting in 2120.

2023-04-07 5.110.2 Adjust future leap seconds starting in 2030. The next leap second is predicted to be a negative leap second on June 30, 2031.

2023-03-24 5.109.2 Adjust future leap seconds starting in 2117.

2023-03-24 5.108.2 Adjust future leap seconds starting in 2093.

2023-03-17 5.107.2 Adjust future leap seconds starting in 2063.

2023-03-10 5.106.2 Adjust future leap seconds starting in 2122.

2023-03-03 5.105.2 Adjust future leap seconds starting in 2030. The next leap second is predicted to be a negative leap second on December 31, 2030.

2023-02-24 5.104.2 Correct a URL. Adjust future leap seconds starting in 2098.

2023-02-17 5.103.2 Adjust future leap seconds starting in 2190.

2023-02-10 5.102.2 Adjust future leap seconds starting in 2029. The next leap second is predicted to be a negative leap second on June 30, 2030.

2023-02-03 5.101.2 Adjust future leap seconds starting in 2229.

2023-01-27 5.100.2 Adjust future leap seconds starting in 2063.

2023-01-13 5.99.2 Adjust future leap seconds starting in 2029. The next leap second is predicted to be a negative leap second on December 31, 2029. Update to IERS Bulletin C 65, issued January 9, 2023, which states that $UTC - TAI$ will remain at -37 seconds for the next six months, at least.

2023-01-11 5.98.2 Move the URLs of the documents from Google Drive to System Eyes Computer Store.

2023-01-06 5.98.2 Adjust future leap seconds starting in 2038.

2022-12-30 5.97.2 Adjust future leap seconds starting in 2169.

2022-12-23 5:96:2 Adjust future leap seconds starting in 2028. The next leap second is predicted to be a negative leap second on June 30, 2029.

2022-12-16 5:95:2 Adjust future leap seconds starting in 2214.

2022-12-09 5:94:2 Adjust future leap seconds starting in 2037.

2022-12-02 5:93:2 Adjust future leap seconds starting in 2256.

2022-11-25 5:92:2 Adjust future leap seconds starting in 2069.

2022-11-18 5:91:2 Adjust future leap seconds starting in 2069.

2022-11-11 5:90:2 Adjust future leap seconds starting in 2028. The next leap second is predicted to be a negative leap second on December 31, 2028.

2022-11-04 5:89:2 Adjust future leap seconds starting in 2197.

2022-10-28 5:88:2 Adjust future leap seconds starting in 2036.

2022-10-21 5:87:2 Adjust future leap seconds starting in 2108.

2022-10-14 5:86:2 Adjust future leap seconds starting in 2027. The next leap second is predicted to be a negative leap second on June 30, 2028.

2022-09-16 5:85:2 Adjust future leap seconds starting in 2027. The next leap second is predicted to be a negative leap second on December 31, 2027.

2022-09-09 5:84:2 Adjust future leap seconds starting in 2135.

2022-08-12 5:83:2 Adjust future leap seconds starting in 2064.

2022-08-05 5:82:2 Adjust future leap seconds starting in 2064.

2022-07-22 5:81:2 Adjust future leap seconds starting in 2036.

2022-07-08 5:80:2 Update to IERS Bulletin C 64, issued July 5, 2022. There will be no leap second at the end of December, 2022. Adjust future leap seconds starting in 2036.

2022-05-27 5:79:2 Adjust future leap seconds starting in 2028. The next leap second is predicted to be a negative leap second on June 30, 2028.

2022-06-10 5:78:2 Adjust future leap seconds starting in 2037.

2022-05-27 5:77:2 Adjust future leap seconds starting in 2028. The next leap second is predicted to be a negative leap second on December 31, 2028.

2022-05-26 5:76:2 Update the URL on the front page.

2022-05-13 5:76:2 Adjust future leap seconds starting in 2028. The next leap second is predicted to be a negative leap second on June 30, 2029.

2022-05-06 5:75:2 Adjust future leap seconds starting in 2028. The next leap second is predicted to be a negative leap second on December 31, 2028.

2022-04-29 5:74.2 Adjust future leap seconds starting in 2038.

2022-04-15 5:73:2 Adjust future leap seconds starting in 2029. The next leap second is predicted to be a negative leap second on June 30, 2029.

2022-04-01 5:72:2 Adjust future leap seconds starting in 2029. The next leap second is predicted to be a negative leap second on December 31, 2029.

2022-03-18 5:71:2 Adjust future leap seconds starting in 2030. The next leap second is predicted to be a negative leap second on June 30, 2030.

2022-03-11 5:70:2 Adjust future leap seconds starting in 2062.

2022-03-08 5:69:2 Try out the EB Garamond font.

2022-02-25 5:69:2 Adjust future leap seconds starting in 2030. The next leap second is predicted to be a negative leap second on December 31, 2030. The Andika font needs more work—I need to find a good monospaced companion. Return to GNU free fonts.

2022-02-21 5:68:2 Try out the Andika font.

2022-02-20 5:68:2 Adjust future leap seconds starting in 2068.

2022-02-19 5:67:2 Correct the date on the front page of the PDF.

2022-02-18 5:67:2 Adjust future leap seconds starting in 2141.

2022-01-07 5:66:2 Adjust future leap seconds starting in 2155. Update to IERS Bulletin C 63, saying there will be no leap second at the end of June, 2022.

2021-12-10 5:65:2 Adjust future leap seconds starting in 2030. The next leap second is predicted to be a negative leap second on June 30, 2031.

2021-11-19 5:64:2 Adjust future leap seconds starting in 2030. The next leap second is predicted to be a negative leap second on December 31, 2030.

2021-11-12 5:63:2 Adjust future leap seconds starting in 2030. The next leap second is predicted to be on June 30, 2031, and it will be a negative leap second.

2021-10-29 5:62:2 Adjust future leap seconds starting in 2068.

2021-10-15 5:61:2 Adjust future leap seconds starting in 2062.

2021-09-26 5:60:2 Move my documents from Wikimedia to Google Drive.

2021-09-24 5:60:2 Adjust future leap seconds starting in 2062.

2021-09-10 5:59:2 Adjust future leap seconds starting in 2072.

2021-08-20 5:58:2 Adjust future leap seconds starting in 2030. The next leap second is predicted to be on December 31, 2030, and it will be a negative leap second.

2021-07-30 5:57:2 Adjust future leap seconds starting in 2076.

2021-07-23 5:56:2 Adjust future leap seconds starting in 2029. The next leap second is predicted to be on June 30, 2030, and it will be a negative leap second.

2021-07-09 5:55:2 Adjust future leap seconds starting in 2412. Update to IERS Bulletin C 62, saying there will be no leap second at the end of December, 2021.

2021-07-02 5:54:2 Adjust future leap seconds starting in 2088.

2021-06-25 5:53:2 Adjust future leap seconds starting in 2091.

2021-06-18 5:52:2 Adjust future leap seconds starting in 2352.

2021-06-11 5:51:2 Adjust future leap seconds starting in 2287.

2021-06-04 5:50:2 Adjust future leap seconds starting in 2063.

2021-05-28 5:49:2 Adjust future leap seconds starting in 2182.

2021-05-21 5:48:2 Adjust future leap seconds starting in 2029. The next leap second is predicted to be on December 31, 2029, and it will be a negative leap second.

2021-05-14 5:47:2 Adjust future leap seconds starting in 2073.

2021-05-07 5:46:2 Adjust future leap seconds starting in 2357.

2021-04-23 5:45:2 Adjust future leap seconds starting in 2029. The next leap second is predicted to be on June 30, 2029, and it will be a negative leap second.

2021-04-16 5:44:2 Adjust future leap seconds starting in 2245.

2021-04-09 5:43:2 Adjust future leap seconds starting in 2063.

2021-04-02 5:42:2 Adjust future leap seconds starting in 2029. The next leap second is predicted to be on December 31, 2029, and it will be a negative leap second.

2021-03-26 5:41:2 Adjust future leap seconds starting in 2030. The next leap second is predicted to be on June 30, 2030, and it will be a negative leap second.

2021-03-19 5:40:2 Adjust future leap seconds starting in 2068.

2021-03-12 5:39:2 Adjust future leap seconds starting in 2030. The next leap second is predicted to be on December 31, 2030, and it will be a negative leap second.

2021-03-05 5:38:2 Include the latest research on historical values of delta T. Adjust essentially all leap seconds before 1825 and in the future. The next leap second is predicted to be on June 30, 2031, and it will be a negative leap second.

2021-03-03 5:37:2 Suggest pdfdetach rather than okular for extracting files from the PDF.

2021-02-26 5:37:2 Adjust future leap seconds starting in 2104.

2021-02-19 5:36:2 Adjust future leap seconds starting in 2051.

2021-02-12 5:35:2 Adjust future leap seconds starting in 2103.

2021-02-05 5:34:2 Adjust future leap seconds starting in 2055.

2021-01-31 5:33:2 Switch back to the Libertine font.

2021-01-29 5:33:2 Adjust future leap seconds starting in 2089.

2021-01-25 5:32:2 Do not send random bits to mktime.

2021-01-22 5:32:2 Adjust future leap seconds starting in 2070.

2021-01-15 5:31:2 Adjust future leap seconds starting in 2039.

2021-01-08 5:30:2 Update to IERS Bulletin C 61, issued January 7, 2021, saying that there will be no leap second at the end of June, 2021. Adjust future leap seconds starting in 2105.

2021-01-01 5:29:2 Adjust future leap seconds starting in 2064.

2020-12-25 5:28:2 Adjust future leap seconds starting in 2077.

2020-12-18 5:27:2 Adjust future leap seconds starting in 2044.

2020-12-04 5:26:2 Adjust future leap seconds starting in 2070.

2020-11-27 5:25:2 Adjust future leap seconds starting in 2097.

2020-11-20 5:24:2 Adjust future leap seconds starting in 2067.

2020-11-15 5:23:2 Update file `fix_files.sh` to include `example_07.py`.

2020-11-13 5:23:2 Convert this document to the Garamond font. Delay the December 2093 leap second by six months to June 2094.

2020-11-06 5:22:2 Adjust future leap seconds starting in 2033.

2020-10-30 5:21:2 Adjust future leap seconds starting in 2033.

2020-10-23 5:20:2 Adjust future leap seconds starting in 2033.

2020-10-16 5:19:2 Adjust future leap seconds starting in 2062.

2020-10-02 5:18:2 Adjust future leap seconds starting in 2059.

2020-09-25 5:17:2 Adjust almost all future leap seconds starting in 2032.

2020-09-04 5:16:2 Adjust almost all future leap seconds.

2020-08-28 5:15:2 Adjust future leap seconds starting in 2089.

2020-08-21 5:14:2 Adjust future leap seconds starting in 2031.

2020-08-14 5:13:2 Adjust future leap seconds starting in 2073.

2020-08-07 5:12:2 Adjust future leap seconds starting in 2114.

2020-07-31 5:11:2 Adjust future leap seconds starting in 2120.

2020-07-24 5:10:2 Adjust future leap seconds starting in 2055.

2020-07-17 5:9:2 Adjust future leap seconds starting in 2080.

2020-07-10 5:8:2 Update to IERS Bulletin C 60, issued July 8, 2020. There will be no leap second at the end of December, 2020. Adjust future leap seconds starting in 2066.

2020-07-03 5:7:2 Adjust many future leap seconds starting in 2025.

2020-06-26 5:6:2 Adjust many future leap seconds starting in 2068.

2020-06-19 5:5:2 Adjust many future leap seconds starting in 2063.

2020-06-12 5:4:2 Adjust many future leap seconds starting in 2098.

2020-06-05 5:3:2 Adjust many future leap seconds starting in 2087.

2020-05-29 5:2:2 Adjust many future leap seconds starting in 2106.

2020-05-21 5:1:2 Adjust several distant future leap seconds.

2020-05-14 5:0:2 Add time_utc_to_foreign_local. Adjust five distant future leap seconds.

2020-05-07 4:15:1 Adjust five distant future leap seconds.

2020-04-30 4:14:1 Convert to Libertine fonts. Adjust several distant future leap seconds.

2020-04-23 4:13:1 Advance three distant future leap seconds by three or six months.

2020-04-16 4:12:1 Advance seven distant future leap seconds by three or six months.

2020-04-15 4:11:1 Recompute all future leap seconds.

2020-04-14 4:10:1 Recompute all future leap seconds.

2020-04-10 4:9:1 Delay the next five leap seconds.

2020-04-03 4:8:1 Update the README file to include Fedora 32.

2020-03-20 4:8:1 Delay the December 2028 leap second by six months to June 2029.

2020-02-21 4:7:1 Delay the December 2023 leap second by six months to June 2024.

2020-02-15 4:6:1 Delay the June 2033 leap second by six months to December 2033.

2020-01-31 4:5:1 Delay the December 2042 leap second by six months to June 2043.

2020-01-25 4:4:1 Delay the June 2023 leap second by six months to December 2023 and the June 2028 leap second by six months to December 2028.

2020-01-11 4:3:1 Update to IERS Bulletin C 59, issued January 7, 2020. There will be no leap second at the end of June 2020.

2020-01-03 4:2:1 Delay the December 2022 leap second by six months to June 2023.

2019-12-20 4:1:1 Delay the December 2032 leap second by six months to June 2033.

2019-12-14 4:0:1 Add `time_test_for_disabled_adjtimex`.

2019-12-13 3:20:0 Delay the December 2027 leap second by six months to June 2028 and the December 2032 leap second by six months to June 2033.

2019-12-06 3:19:0 Delay the June 2022 leap second by six months to December 2022.

2019-12-02 3:18:0 Place the source RPM on github.

2019-11-29 3:17:0 Add a test for adjtimex not working. Mock or container bug reported in Fedora bugzilla 1778298.

2019-11-24 3:16:0 Remove dependency on jdcal, so we can build on more platforms.

2019-11-23 3:16:0 Delay the June 2027 leap second by six months to December 2027.

2019-11-16 3:15:0 Make dependence on 128-bit integers optional. Delay the June 2032 leap second by six months to December 2032.

2019-11-09 3:14:0 Simplify the Makefile and clean up the test programs for rpmbuild. Include a spec file in the tarball for two RPMs: libtime and libtime-devel. Move the source files to their own subdirectory, the example files to theirs, and the test and demo files to theirs.

2019-10-31 3:13:0 Delay the December 2021 leap second by six months to June 2022 and the June 2037 leap second by six months to December 2037.

2019-10-11 3:12:0 Delay the December 2026 leap second by six months to June 2027, and the June 2042 leap second by six months to December 2042. Improve the internal description of Julian Day Number.

2019-10-04 3:11:0 Delay the December 2031 leap second by six months to June 2032.

2019-09-20 3:10:0 Delay the June 2021 leap second by six months to December 2021.

2019-09-07 3:9:0 Delay the June 2026 leap second by six months to December 2026.

2019-08-17 3:8:0 Delay the December 2036 leap second by six months to June 2037.

2019-07-27 3:7:0 Update the leap seconds before 1840 based on the latest research on the rotation of the Earth.

2019-07-04 3:6:0 Update to IERS Bulletin C 58, issued July 4, 2019, which states that no leap second will be introduced at the end of December 2019, so UTC − TAI will remain at −37 seconds for the next six months, at least.

2019-06-22 3:5:0 Be more careful with the definition and display of Julian Day Numbers.

2019-06-16 3:4:0 Delay the June 2020 leap second to June of 2021, and generally revise the leap second schedule by using a new algorithm to predict them.

2019-06-09 3:3:0 Delay the June 2076 leap second by six months to December of 2076.

2019-06-04 3:2:0 Delay the June 2036 leap second by six months to December of 2036. Say more about Python.

2019-05-19 3:1:0 Delay the December 2050 leap second by six months to June of 2051. Add a Python example.

2019-04-28 3:0:0 Add support for Python. Int128_to_string now takes its value argument by reference instead of by value.

2019-04-20 2:11:0 Delay the December 2039 leap second to June of 2040.

2019-04-09 2:10:0 Advance the June 2040 leap second back to December 2039. Show the predictions for the next two leap seconds.

2019-03-30 2:9:0 Delay the December 2039 leap second by six months to June 2040, and delay the June 2065 leap second by six months to December 2065.

2019-03-24 2:8:0 Return the next leap second to June 30, 2020, and adjust some of the leap second predictions for the rest of the twenty-first century.

2019-03-16 2:7:0 Advance the next leap second from June 30, 2020 to December 31, 2019, and adjust the leap second predictions for the rest of the century.

2019-01-20 2:6:0 Advance the June 2095 leap second by six months, returning it to December 2094.

2019-01-12 2:5:0 Update to IERS Bulletin C 57, issued January 7, 2019, which states that no leap second will be introduced at the end of June 2019, so UTC − TAI will remain at −37 seconds for the next six months, at least.

2018-12-23 2:4:0 Delay the December 2094 leap second by six month to June 2095.

2018-12-02 2:3:0 Add information on converting IEEE 1588 (PTP) times to readable text.

2018-12-01 2:2:0 Delay the December 2077 leap second by six months to June 2078.

2018-11-25 2:1:0 Improve the algorithm for anticipating future leap seconds.

2018-11-11 2:0:0 Changed "variable length seconds before 1972" to "variable length seconds before year". This additional flexibility is in honor of Microsoft Windows, which will be counting leap seconds after January 1, 2019.

2018-10-04 1:15:0 Update the list of dependencies in the README file for Fedora 29.

2018-09-30 1:14:0 Improve the prediction of future leap seconds.

2018-09-22 1:13:0 Move the prediction of the next leap second from December 31, 2020 to June 30, 2020 based on predictions of the Earth's rotation by the IERS projected into the future.

2018-07-15 1:12:0 Update to IERS Bulletin C 56, issued July 5, 2018, which states that no leap second will be introduced at the end of December 2018, so $UTC - TAI$ will remain at -37 seconds for the next six months, at least.

2018-05-20 1:11:0 Move the prediction of the next leap second from June 30, 2020 to December 31, 2020 based on predictions of the Earth's rotation by the IERS projected into the future.

2018-05-06 1:10:0 Move the prediction of the next leap second from December 31, 2020 to June 30, 2020 based on predictions of the Earth's rotation from the IERS projected into the future.

2018-01-15 1:9:0 Update to IERS Bulletin C 55, issued January 9, 2018, which states that no leap second will be introduced at the end of June 2018 so $UTC - TAI$ will remain at -37 seconds for the next six months, at least.

2017-12-04 1:8:0 Add information about International Atomic Time.

2017-11-19 1:7:0 Add support for pkg-config, make check and make distcheck. Recommend CLOCK_BOOTTIME rather than CLOCK_MONOTONIC because CLOCK_MONOTONIC stops counting when the computer is suspended, and therefore can change its epoch while an application is running.

2017-10-08 1:6:0 Improve the MAN page.

2017-09-17 1:5:0 Place on github.

2017-08-27 1:4:0 Add a MAN page.

2017-07-08 1:3:0 Use Automake in the build procedure. Distribute the software using the customary GNU packaging, with the PDF file as its documentation. Continue to include the software in the PDF as embedded files. Build the PDF only if requested. Update to IERS Bulletin C 54, issued July 6, 2017, which states that no leap second will be introduced at the end of December, 2017, so $\text{UTC} - \text{TAI}$ will remain at -37 seconds for the next six months, at least.

2017-05-07 1:2:0 Minor typographical changes, add test_local.c, work around compiler over-optimization in int128_to_string.

2017-01-27 1:1:0 Update the Makefile and some of the test programs based on feedback from Steve Summit.

2017-01-18 1:0:0 Added the ability to use variable length seconds before 1972.

2017-01-10 0:2:0 Updated to IERS Bulletin C 53, issued January 9, 2017, which states that $\text{UTC} - \text{TAI}$ will remain at -37 seconds for the next six months, at least.

2017-01-01 0:1:0 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 0:0:0 Original distribution.

References

[1] IEEE. IEEE/Open Group Standard for Information Technology–Portable Operating System Interface (POSIX™) Base Specifications, Issue 8. *IEEE/Open Group Std 1003.1-2024 (Revision of IEEE Std 1003.1-2017)*, pages 1–4107, 2024. doi: 10.1109/IEEESTD.2024.10555529. 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: https://ui.adsabs.harvard.edu/abs/1986PEPI...44..281M. 89

[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. doi: 10.1177/002182860403500305. URL: https://ui.adsabs.harvard.edu/abs/2004JHA....35..327M Provided by the SAO/NASA Astrophysics Data System. 89

[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: https://ui.adsabs.harvard.edu/abs/2005JHA....36..339M Provided by the SAO/NASA Astrophysics Data System. 89

[5] L. V. Morrison, F. R. Stephenson, C. Y. Hohenkerk, and M. Zawilski. Addendum 2020 to 'Measurement of the Earth's rotation: 720 BC to AD 2015'. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 477(2246), Feb 2021. ISSN 1364-5021. doi: 10.1098/rspa.2020.0776. URL: https://royalsocietypublishing.org/doi/10.1098/rspa.2020.0776. 89

[6] John Sauter. Extending Coordinated Universal Time to Dates Before 1972. September 2025. URL: https://www.systemeyescomputerstore.com/leap_-seconds/proleptic_UTC.pdf. 15

[7] 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: https://ui.adsabs.harvard.edu/abs/2011ASSP...23....3S Provided by the SAO/NASA Astrophysics Data System. 89

[8] 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: https://ui.adsabs.harvard.edu/abs/1997A%26A...322..347S Provided by the SAO/NASA Astrophysics Data System. 89

[9] 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 A: Mathematical, Physical and Engineering Sciences*, 472(2196), 2016. ISSN 1471-2946. doi: 10.1098/rspa.2016.0404. URL: https://rspa.royalsocietypublishing.org/content/472/2196/20160404. 89