Leap Years in C for Embedded Systems
Understanding and Calculating Leap Years

Roy Ratcliffe[1,*]

Abstract

This article delves into the intricacies of leap years, exploring their historical context, the rules governing their occurrence, and practical methods for calculating them in embedded systems. We discuss the significance of leap years in maintaining calendar accuracy and provide efficient algorithms suitable for resource-constrained environments. The article also highlights common pitfalls and best practices for implementing leap year calculations in C programming, ensuring reliability and precision in timekeeping applications.

Keywords: Embedded, C

1. The Problem

What is a leap year? In simple mathematical terms,

$$f_{leap}(year) = (year \bmod 4 = 0) \land [(year \bmod 100 \neq 0) \lor (year \bmod 400 = 0)]$$

where $x \bmod y$ is the integer "modulo" operation: the remainder after dividing a numerator by a denominator; the modulo operation computes the "modulus."

Developing a super-simple library for manipulating Gregorian epoch times that account for leap years is the goal of this article. Such a library has no low-level dependencies. It depends only on raw C, making it suitable for embedded systems.

1.1. *Does a Year Leap?*

In basic C, the `is_leap` function answers `true` or `false`:

```c
#include <stdbool.h>

/*!
 * \brief Determine if a year is a leap year.
 * \details Is a year a leap year? A year is a leap year if it is divisible by
 * four, except for years that are divisible by 100, unless they are also
 * divisible by 400.
 * \param year The year to check.
 * \retval \c true if the year is a leap year.
 * \retval \c false if the year is not a leap year.
```

*Corresponding author
Email address: roy@ratcliffe.me (Roy Ratcliffe)
[1]See more hackery at GitHub.

```c
 */
static inline bool is_leap(int year) {
  /*
   * Allow C99's standard precedence to rule over operator ordering: modulo
   * exceeds equality and inequality operators.
   */
  return year % 4 == 0 && (year % 100 != 0 || year % 400 == 0);
}
```

Notice that *year* exists in integer space: $year \in \mathbb{Z}$. This is by design and not just for performance. A year is a whole number for the purpose of leap determination. For contemporary embedded systems, an integer is typically a signed 32-bit number; this provides more than enough resolution for epoch years, which would normally fall between 1970 and 2025, the current year.

2. The Solution

Computing leap years through a range requires a specialised quotient-modulo function that handles negative divisors correctly. The following sections expand on this issue and walk through the complete implementation.

2.1. Leap Through the Years

Another way of looking at the leap year formula exists. For some *year*, the following sum gives the total number of "leap years" *through* the range $thru_{leap} \in [0, year)$.

$$thru_{leap} \in [0, year) = f_{quo}(year, 4) - f_{quo}(year, 100) + f_{quo}(year, 400)$$

Note carefully: the exclusive upper boundary. This is important. The sum does not include the given year; it includes all the years up to the year, but not the year itself. The f_{quo} function computes the integer quotient of its two arguments.

2.1.1. Quotient and modulus

The $f_{quo}(x, y)$ function amounts to, in C:

```c
/*!
 * \brief Quotient and remainder in integer space.
 * \details Structure encapsulating the integer quotient and modulus.
 * The structure is returned by the \c quo_mod() function.
 *
 * This structure mirrors the standard C \c div_t structure returned by the
 * \c div() function, but uses different member names.
 *
 * \see quo_mod()
 */
struct quo_mod {
  /*!
   * \brief Integer quotient.
   */
  int quo;
  /*!
   * \brief Integer modulus.
   */
  int mod;
};

/*!
 * \brief Compute the integer quotient and modulus.
 * \details Performs an integer modulo operation. Then subtracts the modulus from
 * the numerator and applies an integer division in order to compute the
 * quotient.
 *
 * The following invariant proves true: the numerator matches the denominator
 * multiplied by the quotient plus the modulus.
 * \code
 * struct quo_mod qm = quo_mod(x, y);
 * x == (y * qm.quo + qm.mod);
 * \endcode
 * \param x Numerator integer.
 * \param y Denominator integer. Must not be zero.
 * \return \c quo_mod structure comprising the quotient and modulus.
 * \remarks Like Lua's modulo operator, this function ensures that the
 * modulus is always non-negative when the denominator is positive, and always
 * non-positive when the denominator is negative.
 * \note The \c quo_mod identifier exists in structure namespace as well as
 * function namespace.
 * \warning Throws a division-by-zero error if the denominator \c y is zero.
 */
static inline struct quo_mod quo_mod(int x, int y) {
  /*
   * Compute modulus using C's % operator. Note that C's % operator will yield
   * negative results when the numerator is negative.
   */
  int mod = x % y;

  /* Adjust negative modulus to be positive. Ensures that:
   *
   *     0 <= mod < y     when y > 0
   *     y < mod <= 0     when y < 0
   *
   * This matches Lua's modulo operator behaviour.
   */
  if (mod != 0 && (mod ^ y) < 0) {
    mod += y;
  }

  /*
   * Returns a quo_mod structure by casting an initialiser. Is this portable?
   */
  return (struct quo_mod){.quo = (x - mod) / y, .mod = mod};
}
```

The implementation tracks Lua's module operator. The sign differs. In C, the sign of the modulus follows the sign of the numerator, but for Lua, it follows the sign of the divisor. The C implementation needs a small correction; add the divisor if the signs differ between the modulus and the divisor.

$$m' = x \bmod y$$

$$m = \begin{cases} m' + y & \text{if } m' \neq 0 \text{ and } \operatorname{sgn}(m') \neq \operatorname{sgn}(y) \\ m' & \text{otherwise} \end{cases}$$

$$q = \frac{x - m}{y}$$

Properties include:

$$\begin{aligned} 0 \leq m < y & \quad \text{when } y > 0 \\ y < m \leq 0 & \quad \text{when } y < 0 \end{aligned}$$

And the invariant $x = y \cdot q + m$ applies. The simple reason for this disparity between Lua and C boils down to the fact that Lua uses only floating-point numbers at version 5.1. For that reason, it computes `%` in $\mathbb{R}$ space using division and “floor” as follows.

$$\text{luai}_{nummod}(a, b) = a - \left\lfloor \frac{a}{b} \right\rfloor \cdot b$$

2.1.2. Leap through implementation in C

The “leap years through some year” function in C amounts to a simple sum of quotients.

```c
/*!
 * \brief Leap years completed from year 0 up to but not including the first day
 * of the specified year.
 * \details Counts the number of leap years that have occurred from year 0 up to
 * but not including the first day of the given year. This is calculated as the
 * number of years divisible by 4, minus those divisible by 100, plus those
 * divisible by 400. This accounts for the rules of leap years in the Gregorian
 * calendar.
 * \param year The target year up to which leap years should be counted.
 * \returns The total number of leap years from year 0 through the specified
 * year.
 */
static inline int leap_thru(int year) {
  /*
   * Expand the quotient terms first for debugging. Make it easier to see the
   * terms of the thru-sum.
   */
  const int q4 = quo_mod(year, 4).quo;
  const int q100 = quo_mod(year, 100).quo;
  const int q400 = quo_mod(year, 400).quo;
  return q4 - q100 + q400;
}
```

The implementation expands the quotient terms for clarity; they make debugging easier since each term appears separately in the debugger. The following unit tests verify the implementation.

```c
#include "leap.h"

#include <assert.h>
```

```c
void leap_thru_test(void) {
  assert(leap_thru(0) == 0);
  assert(leap_thru(1) == 0);
  assert(leap_thru(2) == 0);
  assert(leap_thru(3) == 0);
  assert(leap_thru(4) == 1);
  assert(leap_thru(5) == 1);
  assert(leap_thru(100) == 24);
  assert(leap_thru(101) == 24);
  assert(leap_thru(200) == 48);
  assert(leap_thru(201) == 48);
  assert(leap_thru(400) == 97);
  assert(leap_thru(401) == 97);
}
```

2.2. *Leap-Adjusted Days and Offset*

Now that the leap library can compute the number of leap years in the range $[0, year)$, the number of days in that range becomes readily derived. Start by assuming 365 days per year *without* adjustment. Apply the leap-through adjustment for all the preceding years. The result is the number of epoch days starting at year 0 and ending at the given *year*.

```c
/*!
 * \brief Counts leap-adjusted days up to some year.
 * \details Counts the number of days completed up to but not including the
 * first day of the year.
 * \param year The year to check.
 * \returns The number of leap-adjust days completed up to but not including the
 * first day of the given year.
 */
static inline int leap_day(int year) {
  return year * 365 + leap_thru(year - 1) + 1;
}
```

The +1 in `leap_day` anchors the epoch so that the start of year 0 maps to day 0 while still counting year 0 as a leap year. The closed-form term `year * 365 + leap_thru(year - 1)` counts 365-day years plus leap days up to (but not including) the target year; because year 0 itself is a leap year in the proleptically[2] applied Gregorian model, that formula alone would put `leap_day(0)` at -1 and undercount every subsequent absolute day number by one. Adding 1 fixes the baseline: `leap_day(0)` becomes 0, `leap_day(1)` becomes 366[3] and differences between years remain correct because the constant offsets cancel in subtractions.

2.2.1. *Leap offset*

The `leap_off` function (see implementation below) normalises an arbitrary day offset relative to a given year into a canonical $(year, day_{of_{year}})$ pair where $0 \leq day_{of_{year}} < days_{in_{year}}$. Its algorithm works as follows:

- Compute the current year's length: `days = 365 + leap_add(year)`.
- While *day* lies outside $[0, days)$:
  - Jump whole years using floor-like quotient semantics: `year0 = year + quo_mod(day, days).quo`.
  - Rebase the offset to the new year using absolute day counts: `day += leap_day(year) - leap_day(year0)`.
  - Update `year` to `year0` and recompute days for that year.

[2]The computations apply the Gregorian model back to Julian time. Pope Gregory XIII's issued the papal bull in 1582.

[3]365 regular days plus the leap day of year 0

- Return the resulting `(struct leap_off){year, day}` which is within the year's bounds.

Notes:

- `quo_mod` uses Lua-style modulo semantics, so negative offsets jump the correct number of whole years in the negative direction.
- Differences of `leap_day(year)` cancel the constant epoch offset, ensuring exact rebasing regardless of the +1 anchor in `leap_day`.

```c
/*!
 * \brief Leap offset by year and day.
 * \details Represents a (year, day-of-year) pair where day-of-year is
 * guaranteed to be within the bounds of the year: 0 <= day < 365 or 366.
 */
struct leap_off {
  /*!
   * \brief Year offset.
   */
  int year;
  /*!
   * \brief Day of year offset.
   */
  int day;
};

/*!
 * \brief Offsets year and day of year.
 * \details Year and day of year from year and day of year. Adjust the day so
 * that it sits in-between 0 and 365 or 366, inclusively.
 *
 * The day adjusts to be within the range of 0 to 365 or 366 inclusively, and
 * the year is adjusted accordingly.
 *
 * This function ensures that the day of the year does not exceed the number of
 * days in the year, accounting for leap years. If the day is negative or
 * exceeds the number of days in the year, it adjusts the year and day
 * accordingly. The day is adjusted to be 0-based, meaning it starts from 0 for
 * the first day of the year. The year is adjusted to account for the number of
 * days in the year, which can be either 365 or 366 depending on whether it is a
 * leap year.
 * \param year The year.
 * \param day_off The day of the year offset, starting from 0. It can be negative
 * or exceed the number of days in the year.
 * \retval leap_off.year The year adjusted to account for the number of days in
 * the year. The resulting year becomes normalised according to the day
 * adjustment whether positive or negative.
 * \retval leap_off.day The adjusted day of the year, starting from 0 for the
 * first day. The day is guaranteed to be within the bounds of the year. 0 <=
 * day < 365 or 366 depending on whether it is a leap year.
 */
static inline struct leap_off leap_off(int year, int day_off) {
  int days = 365 + leap_add(year);
  while (day_off < 0 || day_off >= days) {
    int year0 = year + quo_mod(day_off, days).quo;
    day_off += leap_day(year) - leap_day(year0);
    days = 365 + leap_add(year = year0);
  }
  return (struct leap_off){.year = year, .day = day_off};
}
```

Notice that the `day_off` parameter is zero-based: the first day of the year is day 0. This argument is a day *offset* relative to some year, not a one-based day of the month. This is an important distinction. The day is a zero-based offset in days compared to some base year. The function returns a `struct leap_off` containing the adjusted year and day; also an offset.

This is a key function for manipulating epoch days in embedded systems where no standard library exists. It allows day offsets to be normalised into year-and-day pairs that embedded applications can use for date calculations. Relative leap-year computations become straightforward, as in the following example.

```c
  /*
   * Negative day offset that normalises to the previous year.
   * Year 5 offset -1 day normalises to year 4 day 365 (leap year).
   */
  assert(equal_leap_off((struct leap_off){4, 365}, leap_off(5, -1)));
```

2.3. *Month and Year Day*

Given some *year* and some $1 \leq month \leq 12$ ordinal, computing the days in the month or the days of the year for that month becomes straightforward, and a date (year, month ordinal, day of month ordinal) from a year-day offset conversion falls out.

```c
/*!
 * \brief Day of month from year and month.
 * \details Returns the number of days in the month for the given year and
 * month. Accounts for leap years in February.
 * \param year The year.
 * \param month The month ordinal, starting from 1 for January.
 * \retval The number of days in the month, accounting for leap years in
 * February.
 */
int leap_mday(int year, int month) {
  static const int MDAY[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
  const struct quo_mod qm = quo_mod(month - 1, 12);
  return MDAY[qm.mod] + (qm.mod == 1 ? leap_add(year + qm.quo) : 0);
}

/*!
 * \brief Day of year from year and month.
 * \details Returns the day of the year for the given year and month. The day of
 * the year is calculated by summing the days in the months up to the given
 * month, and adding an extra day if the month is after February in a leap year.
 * \param year The year.
 * \param month The month ordinal, starting from 1 for January.
 * \retval The day of the year, starting from 0 for first of January.
 */
int leap_yday(int year, int month) {
  static const int YDAY[] = {0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334};
  const struct quo_mod qm = quo_mod(month - 1, 12);
  return YDAY[qm.mod] + (qm.mod > 1 ? leap_add(year + qm.quo) : 0);
}
```

2.3.1. Leap Date

The year-day offset to date normalisation function applies the following algorithm for Converting a (year, day-of-year cardinal offset) pair into a (year, month ordinal, day-of-month ordinal) triple.

- Normalise the (year, day) pair using leap_off to ensure day is within the year's bounds.
- Iterate months from 1 to 12, subtracting the number of days in each month from day until day is less than the number of days in the current month.
- The current month is the target month, and day + 1 is the target day of the month (to convert from 0-based to 1-based).

```
/*!
 * \brief Leap year date structure.
 * \details Represents a date in terms of year, month, and day of month,
 * accounting for leap years.
 */
struct leap_date {
  /*!
   * \brief Year.
   */
  int year;
  /*!
   * \brief Month of year starting from 1 for January.
   * \details The month is one-based, starting from 1 for January through 12 for
   * December. The month is therefore an ordinal value in the range 1 to 12
   * inclusive, not a cardinal value starting from 0.
   */
  int month;
  /*!
   * \brief Day of month starting from 1 for the first day of the month.
   * \details The day of the month is one-based, starting from 1 for the first
   * day of the month through to 28, 29, 30, or 31 depending on the month and leap
   * year status.
   */
  int day;
};

/*!
 * \brief Date from year and day of year.
 * \details Adjust the day so that it sits in-between 1 and 365 (or 366) inclusively.
 * Returns the year, month, and day of the month for the given year and day of
 * year. The day of the month is adjusted to be 1-based, meaning it starts from 1
 * for the first day of the month.
 * \param year The year.
 * \param day_off Day of the year offset, starting from 0 for first of January.
 * \retval leap_date.year The year.
 * \retval leap_date.month The month, starting from 1 for January.
 * \retval leap_date.day The day of the month, starting from 1 for the first day
 * of the month.
```

```c
 */
static inline struct leap_date leap_date(int year, int day_off) {
  struct leap_off off = leap_off(year, day_off);
  int month = 1;
  for (; month <= 12; ++month) {
    const int mday = leap_mday(off.year, month);
    if (off.day < mday) {
      break;
    }
    off.day -= mday;
  }
  return (struct leap_date){
      .year = off.year,
      .month = month,
      .day = off.day + 1,
  };
}
```

We can now translate year-day pairs to year-month-day triples.

```c
  /*
   * Three hundred and sixty five days from midnight on 1900-01-01 is 1900-12-31
   * since 1900 is not a leap year.
   */
  assert(equal_leap_date((struct leap_date){1900, 12, 31}, leap_date(1900, 364)));
```

3. Conclusions

The `leapc` (leap years in C) mini-library provides a lightweight, self-contained implementation of leap year calculations suitable for embedded systems with minimal dependencies. The implementation is fast and portable, relying only on basic integer arithmetic and division—no heavy floating-point operations or external libraries required. This makes it ideal for embedded firmware deployments where code footprint and computational efficiency matter.

3.1. Day-Level Resolution

The library operates at day-level granularity, which covers most calendar applications. Extending to timestamps is straightforward: divide a time value in seconds by 86400 (seconds per day) to obtain a quotient and remainder. The remainder gives the time-of-day (seconds since midnight), and the quotient represents the epoch day.

To convert to a specific epoch—such as Unix time (1970-01-01) just subtract `leap_day(1970)` from the quotient. This flexibility is a key strength: the library itself is epoch-agnostic. Callers can anchor to any convenient year: `leap_day(1900)` for the Gregorian epoch, `leap_day(2000)` for Windows NT time, or any other reference point. This design avoids baking a single epoch assumption into the core algorithms.

3.2. Scope and Future Work

The implementation handles positive years and works well for contemporary applications spanning 1970 through 9999. It assumes the proleptically-extended Gregorian calendar (applying modern leap rules backward to year 0). Handling negative years (B.C. dates) or extending beyond year 9999 would require minor adjustments but is not addressed in this version.

The complete implementation, test suite, and usage examples are available on GitHub. For embedded developers seeking a minimal, efficient leap year library, `leapc` offers a practical solution.

The library has limitations: it does not handle negative B.C. years. Nor does it address timezone considerations.

References