# Time, Clock, and Calendar Programming In C

**Eric S. Raymond**
<esr@thyrsus.com>

### Table of Contents



## Motivation

The C/Unix time- and date-handling API is a confusing jungle full of the corpses of failed experiments and various other traps for the unwary, many of them resulting from design decisions that may have been defensible when the originals were written but appear at best puzzling today.

The purpose of this document is to help C programmers develop a clear mental model of how it all works so they can step lightly over the corpses and avoid the traps. In the process it also explains, historically, why things are the way they are - because in this case it's much easier to understand the mess we're in when you know how it got that way.

This document is intended to supplement rather than replace your system manual pages. Accordingly, some details - notably, feature macros you might have to enable for certain functions to be visible to your program - are omitted. Our hope is that this document will enable you to read the relevant Unix manual pages with better understanding and less tendency to feel smothered under details.

## Scope

The functions we will cover are the following:

```
int adjtimex(struct timex *);
int adjtime(const struct timeval *, struct timeval *);
unsigned int alarm(unsigned int);
char *asctime(const struct tm *);
char *asctime_r(const struct tm *, char *);
clock_t clock(void);
int clock_getres(clockid_t, struct timespec *);
int clock_gettime(clockid_t, struct timespec *);
int clock_settime(clockid_t, const struct timespec *);
char *ctime(const time_t *);
char *ctime_r(const time_t *, char *);
char *ctime_rz(timezone_t restrict tz, const time_t *clock, char *buf)
int ftime(struct timeb *tp);
double difftime(time_t, time_t);
struct tm *getdate(const char *);
int gettimeofday(struct timeval *tv, struct timezone *tz);
struct tm *gmtime(const time_t *);
struct tm *gmtime_r(const time_t *, struct tm *);
struct tm *localtime(const time_t *);
struct tm *localtime_r(const time_t *, struct tm *);
struct tm *localtime_rz(timezone_t tz, const time_t * restrict clock, struct
tm *result);
time_t mktime(struct tm *);
time_t mktime_z(timezone_t, struct tm *);
int nanosleep(const struct timespec *, struct timespec *);
int settimeofday(const struct timeval *tv, const struct timezone *tz);
unsigned int sleep(unsigned int);
size_t strftime(char *, size_t, const char *, const struct tm *);
char *strptime(const char *, const char *, struct tm *);
time_t time(time_t *);
time_t timegm(struct tm *tm);
time_t timelocal(struct tm *tm);
int timer_create(clockid_t, struct sigevent *, timer_t *);
int timer_delete(timer_t);
int timer_gettime(timer_t, struct itimerspec *);
int timer_getoverrun(timer_t);
int timer_settime(timer_t, int, const struct itimerspec *, struct itimerspec
*);
void tzset(void);
useconds_t ualarm(useconds_t, useconds_t);
int usleep(useconds_t);
timezone_t tzalloc(const char *zone);
void tzfree(timezone_t restrict tz);
const char *tzgetname(timezone_t restrict tz, int isdst);
long tzgetgmtoff(timezone_t restrict tz, int isdst);
```

Most of these functions are declared in the system header `time.h` and are portable across all modern Unix-like systems. Exceptions to this general rule will be noted.

The following summary table may prove useful as a quick reference:

**Table 1. Summary of functions**

| Function name | Group | Data structure | #include | Standard | Obsoleted | Removed | Replacement |
|---|---|---|---|---|---|---|---|

| Function name | Group | Data structure | #include | Standard | Obsoleted | Removed | Replacement |
|---|---|---|---|---|---|---|---|
| sleep | Delay | int seconds | unistd.h | POSIX.1-1988 | | | |
| usleep | Delay | useconds_t | unistd.h | SUSv1 | POSIX-1.2001 | POSIX.1-2008 | nanosleep |
| nanosleep | Delay | struct timespec | time.h | POSIX.1b-1993 | | | |
| alarm | Delay | int seconds | unistd.h | POSIX.1-1998 | | | |
| ualarm | Delay | useconds_t | unistd.h | SUSv1 | POSIX.1-2001 | POSIX.1-2008 | timer_settime |
| getitimer | Delay | struct itimerval | sys/time.h | SUSv1 | POSIX.1-2008 | | timer_gettime |
| setitimer | Delay | struct itimerval | sys/time.h | POSIX.1b-1993 | POSIX.1-2008 | | timer_settime |
| timer_create | Delay | timer_t | time.h | POSIX.1b-1993 | | | |
| timer_delete | Delay | timer_t | time.h | POSIX.1b-1993 | | | |
| timer_getoverrun | Delay | timer_t | time.h | POSIX.1b-1993 | | | |
| timer_gettime | Delay | timer_t | time.h | POSIX.1b-1993 | | | |
| timer_settime | Delay | timer_t | time.h | POSIX.1b-1993 | | | |
| timerfd_create | Delay | | sys/timerfd.h | Linux only | | | |
| timerfd_settime | Delay | struct itimerspec | sys/timerfd.h | Linux only | | | |
| timerfd_gettime | Delay | struct itimerspec | sys/timerfd.h | Linux only | | | |
| time | Clock | time_t | time.h | POSIX.1-1988 | | | |
| difftime | Clock | time_t | time.h | C89 | | | |
| clock | Clock | clock_t | time.h | C89 | | | |
| gettimeofday | Clock | struct timeval | sys/time.h | SUSv1 | POSIX.1-2008 | | clock_gettime |
| settimeofday | Clock | struct timeval | sys/time.h | 4.3BSD/SVr4 | | | clock_settime |
| clock_gettime | Clock | struct timespec | time.h | POSIX.1b-1993 | | | |
| clock_settime | Clock | struct timespec | time.h | POSIX.1b-1993 | | | |

| Function name | Group | Data structure | #include | Standard | Obsoleted | Removed | Replacement |
|---|---|---|---|---|---|---|---|
| clock_getres | Clock | struct timespec | time.h | POSIX.1b-1993 | | | |
| tzset | Date | | time.h | POSIX.1-1998 | | | |
| ftime | Date | struct timeb | time.h | SUSv1 | POSIX.1-2001 | POSIX.1-2008 | clock_gettime |
| gmtime | Date | struct tm, time_t | time.h | POSIX.1-1988 | | | |
| gmtime_r | Date | struct tm, time_t | time.h | POSIX.1c-1995 | | | |
| gmtime_rz | Date | struct tm, time_t,timezone_t | time.h | 4.xBSD | | | |
| localtime | Date | struct tm, time_t | time.h | POSIX.1-1988 | | | |
| localtime_r | Date | struct tm, time_t | time.h | POSIX.1c-1995 | | | |
| gmtime_rz | Date | struct tm, time_t,timezone_t | time.h | 4.xBSD | | | |
| mktime | Date | struct tm | time.h | POSIX.1-1988 | | | |
| mktime_rz | Date | struct tm, time_t,timezone_t | time.h | 4.xBSD | | | |
| timelocal | Date | struct tm | time.h | GNU only | | | |
| timegm | Date | struct tm | time.h | GNU only | | | |
| asctime | Date | struct tm | time.h | POSIX.1-1988 | POSIX.1-2008 | | strftime |
| asctime_r | Date | struct tm | time.h | POSIX.1c-1995 | POSIX.1-2008 | | strftime |
| ctime | Date | time_t | time.h | POSIX.1-1988 | POSIX.1-2008 | | strftime |
| ctime_r | Date | time_t | time.h | POSIX.1c-1995 | POSIX.1-2008 | | strftime |
| strftime | Date | struct tm | time.h | POSIX.1-1988 | | | |
| strptime | Date | struct tm | time.h | SUSv1 | | | |
| getdate | Date | struct tm | time.h | SUSv1 | | | |
| getdate_r | Date | struct tm | time.h | GNU only | | | |
| tzalloc | Date | timezone_t | time.h | 4.xBSD | | | |

| Function name | Group | Data structure | #include | Standard | Obsoleted | Removed | Replacement |
|---|---|---|---|---|---|---|---|
| tzfree | Date | timezone_t | time.h | 4.xBSD | | | |
| tzgetname | Date | timezone_t | time.h | 4.xBSD | | | |
| tzgetgmtoff | Date | timezone_t | time.h | 4.xBSD | | | |

The "Standards" column shows the oldest standard that describes the call, otherwise the earliest (non-standardized) implementation. Some relevant abbreviations:

**Table 2. Standards and origins**

| | |
|---|---|
| C89 | ANSI X3.159-1989 "Programming Language C." |
| POSIX | IEEE 1003, aka ISO/IEC 9945. Development began in 1988, with significant revisions in 1996, 2001, and 2008. |
| SUS | Single UNIX Specification. Merged with POSIX in 2001. |
| GNU | Unstandardized calls in the very widely used GNU C Library. |
| 4.xBSD | The 4.2 (1983) and 4.3 (1986) versions of BSD Unix invented some API elements still in occasional use. |
| SVr4 | System V Release 4 (1988). Basis for later Unix standards. |

A manual page reference with the section part "(2|3)" means you may find pages with the identified name in both sections 2 or 3; usually this happens because there is both native system-call documentation in section 2 and a POSIX page in Section 3.

## Unix time and UTC/GMT/Zulu

Internally, Unix time is represented as SI (*Système International*) seconds since midnight of January 1st 1970 at the Greenwich meridian, without leap-second correction. This is time counted in seconds as though it had been incremented every second in constant-length days of 86400 seconds each since then.

This is sometimes described as "UTC time" or "UTC seconds" because it is based on the same zero meridian and uses the same SI second as Coordinated Universal Time (abbreviated UTC to be neutral between this and the French 'Temps Universel Coordonné'[1]), the modern standard [UTC].

The detailed history of UTC and its antecedents is out of scope for this document (a summary is as [TIMESCALES]), but a few points about it are relevant. One is that leap seconds were not introduced into UTC until 1972. Another is that UTC was then and still is occasionally confused with a previous international time

standard, Greenwich Mean Time (GMT) [GMT]. This is why one of the principal Unix time functions is named `gmtime(3)`.

There is one subtle but important difference between Unix UTC and the official UTC standard time. Official UTC time is by definition solar calendar time (year/month/day/hour/minute/second) rather than a seconds counter from an epoch like Unix UTC time. The difference becomes, as we shall shortly see, significant near leap seconds. But the relationship with solar UTC time remains close, and in the rest of this document we shall continue to speak of Unix UTC.

In U.S. military usage dating back to WWII GMT/UTC is referred to as "Zulu" time; the word "Zulu" means nothing in this context but was chosen to abbreviate to "Z" in order to avoid colliding with any existing timezone designation. Use of the "Zulu" designation spread to international aviation and has left a mark on the ISO-8601 international standard for representing calendar time and date, which uses Z as a suffix to indicate UTC rather than local time. Thus, the Unix epoch (zero second) is formally represented as 1970-01-01T00:00:00Z.

The standard Unix type for holding this UTC/GMT/Zulu seconds counter is `time_t`. It is integral and signed, and negative values designate seconds *before* 1970-01-01T00:00:00Z. The current `time_t` value is returned by the `time(3)` function. (Note that ANSI C allows for `time_t` to be a float value, but this choice seems never to have been made in a real operating system [2] and is excluded in later POSIX revisions. There have been a few approximately Unix-like systems on which `time_t` was unsigned and could not represent dates before 1970, notably QNX.)

The absence of leap-second correction means that occasionally, in order to stay synchronized with solar time, the Unix time counter has to skip or duplicate a second when it crosses UTC midnight during a leap-second insertion or deletion. A very detailed explanation of these discontinuities can be found at the Wikipedia page on Unix time [UNIX-TIME]. The rationale for excluding leap-seconds can be found at [POSIX-TIME].footnote[Note however that one major assertion in [POSIX-TIME], "most systems are probably not synchronized to any standard time", is now incorrect; synchronization to Internet time via NTP became routine within a few years after it was written in 1996.]

In practice, almost the only way these discontinuities could be an issue is if you are computing with differences in timestamps taken on opposite sides of one or more leap-second insertion or deletions. In that case actual elapsed time could be a second or more different than the result of subtracting the older Unix timestamp from the newer.

## Unix time and NTP correction

Most Unix systems rely on Network Time Protocol servers to correct their timebase so that the computer's top-of-second closely matches that of atomic-clock UTC time - usually as defined by the U.S. Naval observatory, but there are other national time authorities which all try to stay in synchronization with each other.

Periodic NTP adjustments are required because the clock oscillators in individual computers are subject to frequency drift due to thermal and other physical effects.

A detailed description of the adjustment process is beyond the scope of this document. The main thing for programmers to know is that NTP almost always tells your local clock to resynchronize by minutely speeding it up or slowing it down until it matches NTP time (rather than skipping or inserting clock increments). In rare situations, like after prolonged network outages or CPU sleep periods, NTP may skip or repeat clock increments.

Thus, in the presence of NTP correction, Unix seconds are of slightly variable width in real time. The maximum divergence is very small (at most a few microseconds) and the typical variation is smaller than that.

Thus, the variability should have no effect except in hard real-time situations. In those, you shouldn't be using an NTP-corrected clock but a native monotonic timer designed for real-time use; we'll see later that the time API provides flexible ways to do this.

Unix kernels have provided NTP with a system call to use in adjusting clock frequency since 4.3BSD; it is `adjtime(3)`. This call is not formally standardized. There is a native Linux variant, `adjtimex(2)`. These are both intended solely for NTP's use, and that is all we will have to say about them in this document.

## Unix time and machine word length

During the evolution of the Unix time API from 1968 onwards, typical machine word lengths have changed twice - from 16 to 32 and then to 64 bits, which is typical today (in 2017) and seems unlikely to change in the foreseeable future.

The changes in word lengths have left some scars in the Unix API. At the very beginning `time_t` was defined as a signed 32-bit quantity, so it couldn't be held in the 18-bit registers of the PDP-7 or the 16-bit registers of the later PDP-11. This is why many of the Unix time calls take a pointer to `time_t` rather than a `time_t`; by passing the address of a 32-bit span in memory the design could get around the narrowness of the register width. This interface glitch was not fixed when word lengths went to 32 bits.

A more serious problem is that 32-bit Unix `time_t` counters will turn over just after 2038-01-19T03:14:07Z. This is expected to cause problems for embedded Unix systems; it is difficult to anticipate their magnitude, but we can only hope the event will be the same sort of damp squib that the Year 2000 rollover turned out to be.

Modern Unix systems use a signed 64-bit `time_t`. These counters will turn over approximately 292 billion years from now, at 15:30:08 on Sunday, 4 December 292,277,026,596. No problems with this are presently anticipated.

Register length limits have also affected the representation of time at subsecond precision. As a workaround against them, and to avoid floating-point roundoff and comparison issues, the C API traditionally avoided representing fractional-second times as a scalar float or double quantity. The reason had to do with the precision offered by different float formats:

**Table 3. Float precision**

| Word size | Mantissa | Exponent | Historical name |
|---|---|---|---|
| 32-bit float | 23 | 8 | Single precision |
| 64-bit float | 52 | 11 | Double precision |
| 128-bit float | 112 | 15 | Quad precision |

(The sums look off-by-one because of the sign bit. You can learn more about the IEEE754 floating-point formats that give rise to these numbers at [FP]. They originated on the VAXen that were the workhorse machines of Unix in the early 1980s and are now implemented in hardware on Intel and ARM architectures, among many other places.)

When the Unix time API first had to represent subsecond precision, microsecond resolution was required to represent times comparable to a machine cycle.

The problem was that a microsecond count requires 20 bits. A microsecond-precision time with 32 bits of integer part is on the far edge of what a double-precision float can hold:

```
seconds:       32 bits
microseconds:  20 bits
               --------
total:         52 bits
```

That would barely have fit and seemed likely to be a bit flaky in actual use due to floating point rounding. Doing any math with it would lose precision quickly. Trying to go finer-grained to nanosecond resolution would have required 11 more bits that weren't there in double precision.

Thus, quad-precision floating point would have been required for even 32-bit times. Given the high cost of FPU computation at the time and the near-waste of 64 bits of expensive storage, this took float representation out of the running.

This is why fractional times are normally represented by two-element structures in which the first member is seconds since the epoch and the second is an integral offset in sub-second units - originally microseconds.

The original subsecond-precision time structure was associated with the `gettimeofday(2)` system call in 4.2BSD Unix, dating from the 1980s. It looks like this:

```
struct timeval {
    time_t      tv_sec;     /* seconds */
    suseconds_t tv_usec;    /* microseconds */
};
```

Note the microsecond resolution. The newer POSIX time functions use this:

```
struct timespec {
    time_t      tv_sec;     /* seconds */
    long        tv_nsec;    /* nanoseconds */
};
```

(No, that's not a paste error. The struct timespec members really do have tv_ prefixes on their names. This seems to have been someone's attempt to reduces required code changes. It was probably a bad idea.)

This has nanosecond resolution. The change is related to the tremendous increase in machine speeds since the 1980s, and the correspondingly increased resolution of hardware clocks. While it is conceivable that in the future we may see further generations of these structures in which the subsecond offset is in picoseconds or smaller units, some breakthrough in fundamental physics would be required first - at time of writing in 2014 processor cycle times seem to be topping out in the roughly 0.1ns range due to quantum-mechanical limits on the construction of electron logic.

Although the timeval and timespec structures are very useful for manipulating high-precision timestamps, there are unfortunately no standard functions for performing even the most basic arithmetic on them, so you're often left to roll your own.

Another structure, used for interval timers and describing a time interval with nanosecond precision, looks like this:

```
struct itimerspec
{
    struct timespec it_interval;
    struct timespec it_value;
};
```

While the C time API tends to shape the time APIs presented by higher-level languages implemented in C, these subsecond-precision structures are one area where signs of revolt are visible. Python has chosen to instead accept the minor problems of using a floating-point scalar representation; Ruby uses integral nanoseconds since the Unix epoch. Perl uses a mixture, a BSD-like seconds/microseconds pair in some functions and floating-point time since the Unix epoch in others.

On today's true 64-bit machines with relatively inexpensive floating point the natural float representation of time would look like this:

```
seconds:            64 bits
fractional seconds: 48 bits
                    --------
total:              112 bits
```

offering sub-picosecond resolution with plenty of headroom to avoid serious roundoff issues. So the scripting languages are heading in a direction that the C API could in theory eventually follow.

## Delay and interval-timing functions

The simplest portions of the Unix time API are clock and delay functions that only deal with the basic second- and fractional-second-oriented time structures.

First we'll survey the delay and interval-timing functions, then the clock functions. If you are mainly interested in clock/calendar/timezone issues, you can safely skip this section.

## sleep(3), usleep(3), and nanosleep(3)

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
int usleep(useconds_t usec);

#include <time.h>

int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
```

These functions are the poster children for the mess created by having multiple time representations. All three simply delay execution of the calling program for some span of time; what differs is how that time is specified.

The oldest, `sleep(3)`, takes an integer argument in seconds; `usleep(3)` takes microseconds, and `nanosleep(3)` takes a `struct timespec` with nanosecond resolution. All three may be interrupted by an unignored signal. The `nanosleep(2)` function takes a second timespec which is filled with time remaining if it was interrupted.

The `usleep(3)` call originated in BSD Unix and, though it was carried forward in POSIX, was deprecated in favor of `nanosleep(2)` in POSIX.1-2001/SUSv3 and then removed entirely in POSIX.1-2008/SUSv4. It's probably best not to count on it being portable.

Of these three functions, only `nanosleep(3)` is guaranteed not to have any interaction with signals (the other two may be implemented using some signal and a hidden handler). It is is thread- and signal-safe.

The `sleep(3)` function is required by POSIX to be async-signal-safe and thread-safe. The thread-safe requirement means `sleep(3)` can't be signal-based in multithreaded programs, but can be in single-threaded programs.

These functions should not be mixed with the ones in the next group.

## alarm(2|3), ualarm(3), getitimer(2|3), and setitimer(2|3)

```
#include <unistd.h>

unsigned alarm(unsigned seconds);
useconds_t ualarm(useconds_t usecs, useconds_t interval);

#include <sys/time.h>

int getitimer(int which, struct itimerval *value);
int setitimer(int which, const struct itimerval *restrict value,
              struct itimerval *restrict ovalue);
```

The alarm functions are another case of call proliferation due to multiple time units. The oldest, `alarm(2|3)`, sends SIGALRM after a specified number of seconds; `ualarm(3)` sends SIGALRM after a specified number of microseconds (and a second argument, if nonzero, causes SIGALRM to be sent at regular intervals afterwards).

The `getitimer()` and `setitimer()` functions come from the Single UNIX Specification before 1996, when SUSv2 was merged with POSIX.1-1996 to develop POSIX.1-2001/SUSv3. They are now deprecated in favor of the POSIX timer_* calls in the next section.

These have three advantages over the traditional alarm calls: (1) they use timeval structures, so have microsecond resolution; (2) they can trigger the sending not just of SIGALRM but of more specialized profiling signals; and (3) they give you options about what derivative of the system hardware clock you want to use.

The gory details about the interval timer functions are best learned from their manual pages. For our purposes, the important thing to cover is how they fit with the rest of the time and calendar API. Bluntly, that is not very well.

The manual pages are full of ominous "unspecified behavior" warnings if you mix uses of the sleep group, the alarm group, and the itimer group in the same program. The reason for this is historical.

The sleep and alarm function groups developed by accretion in early Unixes. The `sleep(3)` group was often implemented with `alarm(3)` and a specialized SIGALRM handler; likewise for `usleep(3)` and `ualarm(3)`.

The itimer functions were the result of a later effort to specify a more general facility that could subsume these, starting from a clean sheet of paper. They were declared obsolete in POSIX.1-2001/SUSv3 in favor of the timer_* group.

Thus, it is possible that your system's implementation of sleep and alarm functions is a thin layer over POSIX itimer calls (or, possibly, the timer_* group documented next) using the ITIMER_REAL clock. When that isn't the case, legacy implementations of the sleep and alarm calls won't necessarily play well with interval timers - in particular SIGALRMs might be flying around when you don't expect it, or setting an implied signal handler through the sleep functions might be interfered with by `setitimer(2|3)`.

It's best not to go there. Heed the warnings and don't mix up these function groups.

## The POSIX timer_* group

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

int timer_create(clockid_t clockid, struct sigevent *restrict evp,
                 timer_t *restrict timerid);
int timer_delete(timer_t timerid);
int timer_getoverrun(timer_t timerid);
int timer_gettime(timer_t timerid, struct itimerspec *value);
int timer_settime(timer_t timerid, int flags,
                  const struct itimerspec *restrict value,
                  struct itimerspec *restrict ovalue);
```

There is a group of POSIX functions `timer_create(2|3)`, `timer_delete(2|3)`, `timer_settime(2|3)`, `timer_gettime(2|3)`, and `timer_getoverrun(2|3)` that can be used to set per-process interval timers even more flexibly than the itimer group. In particular, they allow setting a function-call hook to be called directly at the end of an interval without going through either an explicit or implicit signal.

These functions were introduced in POSIX.1b-1993. After POSIX and SUSv2 merged in 2001 `getitimer(2|3)` and `setitimer(2|3)` were declared obsolete.

Neither the native Linux nor POSIX man pages document interactions with the older calls. A prudent programmer should assume that the alarm and sleep group is implemented in terms of one of the two groups of newer POSIX interval-timer functions, and not mix any of them.

If you want to avoid interval variability due to NTP frequency adjustments, you probably want to use these with CLOCK_MONOTONIC.

## The Linux timerfd_* group

```
#include <sys/timerfd.h>

int timerfd_create(int clockid, int flags);
int timerfd_settime(int fd, int flags,
                    const struct itimerspec *new_value,
```

```
                struct itimerspec *old_value);
int timerfd_gettime(int fd, struct itimerspec *curr_value);
```

These functions are Linux-specific (not standardized), offering a way to monitor timer expiration via selecting or polling on a file descriptor rather than via signal handler.

They are mentioned here for completeness, but should be avoided in code intended for portability. Consult the Linux manual pages for details.

### Other delay-related functions

A full discussion of the `select(2)` and `pselect(2)` system calls would be beyond the scope of this document, as they are not used for time/calendar programming. We mention them only to note two points:

1. `select(2)` takes a `struct timeval` argument, reflecting its origins in BSD Unix. The resolution to which you can specify a timeout is thus limited to a microsecond.
2. The later `pselect(2)` call takes a `struct timespec`.

## Clock and time-of-day functions

These are functions for interacting with the system clock that avoid timezone issues by returning time in seconds since the Unix epoch.

### time(3)

```
#include <time.h>

time_t time(time_t *tloc);
```

The most basic function is `time(3)`, which returns the current Unix second. All the previous caveats about leap-second discontinuities and tiny variabilities due to NTP correction apply.

### difftime(3)

```
#include <time.h>

double difftime(time_t time1, time_t time0);
```

The function `difftime(3)` attempts to return the elapsed time between two `time_t` counters as a double - however, it is not aware of leap-second discontinuities (a fact the man page does not document!).

This call was added in C89 as a way to encapsulate time arithmetic on operating systems where `time_t` is not a normally encoded integral or float type; it is possible (even likely) that no such environments actually exist.

While on POSIX systems where `time_t` is required to be an integral type holding a count of seconds it might appear equivalent to just subtracting the first argument from the second, it is not quite superfluous. It is defined to handle correctly some edge cases where naive subtraction would cause an integer overflow.

### clock(3)

```
#include <time.h>

clock_t clock(void);
```

We mention the function `clock(3)` here only for completeness, because it is declared in `time.h`. It is little used, returning an approximation of processor time used by the running program. This return is **not** a `time_t`, but an integer in microseconds * CLOCKS_PER_SEC (the latter being a constant in <time.h>)

This function is very portable, but you probably want the finer-grained statistics from `getrusage(2)` instead.

### gettimeofday(2|3)/settimeofday(2)

```
#include <sys/time.h>

int gettimeofday(struct timeval *restrict tp, void *restrict tzp);
int settimeofday(const struct timeval *tv, const struct timezone *tz);
```

This is the two oldest Unix interface that deals with subsecond time, dating from 4.2BSD (the other being `ftime(3)`). With it you can get time since the epoch to microsecond precision (it uses a `struct timeval`, not the newer `struct timespec`), or (with root permissions) set the system clock to microsecond precision.

Though restricted versions were carried forward in POSIX, these calls are obsolete, and are documented here mainly so you'll understand how they fit in when reading old code. It is best to use `clock_gettime(2)` and `clock_settime(2)` (which support nanosecond precision) instead of these.

(Some systems may still require you to use `gettimeofday(3)`; notably this is true of Mac OS X up to and including 10.9.5 Mavericks, which conforms to POSIX.1-2001/SUSv3 rather than the POSIX.1-2008/SuSv4 in which `clock_gettime(3)` became mandatory.)

In the POSIX documentation, behavior if the `gettimeofday(3)` historical second argument is non-null is undefined. The Linux native `gettimeofday(2)` (and possibly other implementations) attempts to preserve more of the original BSD semantics, returning some timezone-related information; however, this feature is marked obsolete and should not be relied on.

The `settimeofday(2)` call was never standardized at all, though it was carried forward in System V Release 4 and de-facto standard for some time.

### clock_gettime(2), clock_settime(2), clock_getres(2)

```
#include <time.h>

int clock_getres(clockid_t clk_id, struct timespec *res);
int clock_gettime(clockid_t clk_id, struct timespec *tp);
int clock_settime(clockid_t clk_id, const struct timespec *tp);
```

These are the modern, POSIX-standardized functions for interacting with the system clock at subsecond precision. They use the nanosecond-precision `struct timespec` and allow working with multiple clocks.

The most interesting (and only portable) clocks are identified as CLOCK_REALTIME and CLOCK_MONOTONIC. CLOCK_REALTIME is the familiar UNIX time since the epoch, with leapsecond discontinuities and NTP adjustments. CLOCK_MONOTONIC is a monotonically increasing nanoseconds and seconds counter that cannot be set; note that unlike an interval timer it **is** subject to leapsecond discontinuities and NTP adjustments.

The call `clock_getres(2)` is available to query the actual resolution of the system clock.

## Date and timezone-aware functions

Now we come to the part that is the most interesting to application programmers and (probably) the most confusing. Unfortunately, most of the difficulties here cannot readily be solved in software.

The calendaring parts are not problematic. The Unix time API supports the Gregorian calendar, which is well standardized and now in sufficiently universal use worldwide that any conversion problems to local standards can be handled locally.

The real difficulties arise from timezones, especially when a program must deal with or report times from a zone other than the one configured as local on the machine where it is running.

## The tragedy of timezones

Before the mid-19th century, clocks were set (more or less haphazardly) to indicate local solar time. The goal was essentially for noon to coincide with the average time of the Sun's maximum declination. Humans moving by foot, horse, or ship did not move quickly enough to make time synchronization between regions with different mean solar times a problem (with a very limited technical exception for marine navigation).

The impetus for standard time came from rail travel. The railroads needed standardized time in order to set and publish precise schedules [3]. While there were significant commercial advantages to adopting railroad time as a civil standard wherever it reached, human beings did not want to give up rough synchronization of their clocks and watches to local mean solar time.

Time zones developed as a compromise. By partitioning the globe into hour-wide longitudinal bands with a fixed time offset from one world reference time, two desirable properties could be achieved. First, rough synchronization with mean solar time would be maintained everywhere. Second, the relationship between standard time and any local time would remain easy to compute.

The actual time zone system never approximated this ideal very closely. Adoption was slow and patchy after the development of the first railroad standard time system (in 1847 in Great Britain) and was not roughly complete until the early 20th century.

From the beginning, various time authorities showed a tendency to move timezone borders to avoid putting regions they were closely tied to off into a different timezone, even if that meant accepting large clock deviations from mean solar time in outlying regions. Some regions adopted half-hour or even quarter-hour offsets. These meant, unhappily for computer programs and programmers, that the time offset of a location is not a simple function of its longitude.

Worse, the time zone system has been unstable over time as the political and commercial pressures on it resulted in frequent changes to the definitions of zones. And still worse than that was the partial and unstable adoption of Daylight Saving Time, aka DST, aka Summer Time, in which the time definition within a zone endures further changes, by another hour twice a year, so people get up an hour earlier and thus get an extra hour of daylight in the evening.

As a final wrinkle, some jurisdictions use different timezone names when DST is in effect; this is common in the U.S. where, for example, the Eastern timezone uses Eastern Standard Time (EST) as its basic designator but Eastern Daylight Time (EDT) during summer months. But this is not necessarily true everywhere.

More historical details can be found at [TIME-ZONE]. The result of this mess is the following:

1. Every computer has to be configured with (at least) a local time offset - that is, the difference between its local time and UTC.
2. In order to display and/or interpret local times from other (named) timezones, any computer that wants to do it needs a database of named-timezone-to-offset mappings and DST start/end times for each zone. Each zone may in fact require a **series** of such mappings spanning different historical and future ranges of UTC time; the database needs to be updated whenever a zone changes its rules.
3. There was a lot of pressure from early on in the development of software standards to abandon named timezones in computer date formats in favor of transmitting dates as local time plus the originating location's numeric offset from meridian time, usually in the form **+hhmm** or **-hhmm**. While this still required every computer to configure the local offset (and perhaps change it twice a year for DST) it at least removed the requirement for a timezone database.
4. Past timestamps using named time zones have to be interpreted with caution, in particular because it may not be evident whether DST was in effect or not.
5. Embedded systems often evade the whole issue by wiring themselves to Zulu time no matter where they're deployed.

For an entertaining video lecture about the messy consequences of this history, see [COMPUTERPHILE]. As he says near the end, your only sane option (assuming you don't go the everything-is-UTC route) is to trust that tzdata or whatever other timezone history your operating system subscribes to is properly maintained and then simply refuse to worry about that level of the problem, because you'd go mad if you tried.

## Timezone selection and representation on Unix

In the earliest Unixes, a machine could handle exactly one timezone other than UTC. A timezone offset and name string (actually, a pair of name strings, one for summer and one for winter time) were configured into the kernel and available to C programs.

Later (by System III in 1982) it became possible to set a login session's notion of the local time zone by modifying an environment variable named TZ and calling a function `tzset(3)`. The rules for interpreting the zone specification in the value of TZ became part of the POSIX standard. They were complex [OLDTZ] and now mainly of historical interest. We mention this history mainly because it has left remnants in the manual pages - notably of `tzset(3)` - that may confuse the uninformed reader about what is going on.

V7 and older BSD Unixes had various other methods for configuring the local timezone that didn't involve using or interpreting TZ. These have left no traces on modern Unixes. They shared a fatal flaw with POSIX TZ interpretation, which was that they weren't designed to cope with the historical instability of the timezone system. They could not express an entire set of historical offset/DST rules in order to get past local times as well as present ones correct.

On modern Unix systems the TZ variable might not be set at all, but the system default timezone can be overridden by explicitly setting TZ in any process. The timezone designator configured at boot time, or via an overriding value of TZ, is a geographical location (usually but not always an area/major-city pair) such as "America/New_York", or "Europe/Vienna" or "Asia/Taipei". If the designator is set via TZ it may need to be preceded by a colon; this is for backwards-compatibility with the POSIX standard, to distinguish it from an old-style timezone specification. (Not all implementations enforce this.)

The location-based zone naming scheme [IANA-ZONES] is managed by IANA, the Internet Assigned Numbers Authority. It is intended to supersede (in computing, anyway) the system of customary names we'll refer to later in this document as "civil timezone names".

Civil timezone names such as EST/EDT are ambiguous: they have name collisions in different countries, and some civil zones have multiple names. They tend to be short (at least under Unix/Linux, though not necessarily under Windows). They may change without notice. Some of them are official (in that they appear in officially-approved legislative documents in the jurisdictions where they apply), but many of them reflect mere ad-hoc conventions.

The IANA names, on the other hand, are intended to be unique, future-proof, and for use by system administrators and knowledgeable users to **set** the time zone they wish to apply.

IANA timezone designators are looked up through a locally-installed copy [4] of a time zone database ("tzdata") maintained by IANA, the Internet Assigned Numbers Authority [TZSOURCES], [TZDATA]. You may hear the name "Arthur David Olson" used in connection with this database; he was the founding contributor.

The contents of the database entry for the designator describe the history of the zone name, zone offset and the DST start/end times (if any) for the location, as those have changed over time since 1970 [5]. The timezone system is sufficiently chaotic in ways previously noted that some of the comments on this history make entertaining reading.

## Unix date formats

The Unix time API evolved when many system designers and software standardizers were still trying to hang on to named timezones, which were thought to be more human-friendly. Internet protocols evolving around the same time, however, abandoned named timezones early.

The results of this history are visible as a plethora of different date presentation formats. Thus, for example, the output of the `date(1)` command looks like this: "Wed Sep 24 15:32:27 EDT 2014" (with named timezone), while a date in an SMTP mail header is more likely to look like this: "Wed, 24 Sep 2014 15:32:27 -0400" with an **hhmm** numeric offset.

Here is a table duplicating this example in a number of presentation formats commonly found on Unix systems:

**Table 4. Date formats**

| | |
|---|---|
| 1414179147 | Unix UTC seconds |
| Fri Oct 24 15:32:27 EDT 2014 | `date(1)` output |
| 2014-10-24 15:32:27-0400 | `date(1)` output with -rfc-3339 option |
| Fri, 24 Oct 2014 15:32:27 -0400 | e-mail RFC-822/RFC-2822 format |
| Fri, 24 Oct 2014 19:32:27 GMT | HTTP (RFC-2616/RFC-7231) format |
| 20141024192327.000000Z | LDAP (RFC-2252/X.680/X.208) format |
| 2014-10-24 15:32:27 | Modified ISO-8601 local time |
| 2014-10-24T15:32:27 | Strict ISO-8601 local time |
| 2014-10-24T19:32:27Z | RFC-3339 time, always UTC and marked Z |

Of these, the last is the most recent and (so far) least widely adopted presentation format; ISO-8601 was not promulgated until 1988 and RFC-3339 not published until 2002 [ISO-8601], [RFC-3339]. However, it has the best properties of any (human readability, non-ambiguity, compactness, fixed length, single token, string sort order matches time order) and we strongly recommend it.

All of these presentation formats are straightforwardly constructible using the strftime(3) function which we'll be meeting in a few paragraphs (although parsing them can be trickier).

(See also Randall Munroe's ISO 8601 and Supervillain Plan.)

Some Unix time API functions deal in numeric zone offsets, others in named timezone and DST. Still others (generally older ones) evade the whole issue by not reporting timezone at all. We will be more specific when we discuss individual calls.

Just to add to the hilarity, the IANA database, RFC-822/RFC-2822 dates, and Unix `date(1)` consider offsets east of Greenwich to be positive and west of it to be negative, but POSIX time-zone formats inexplicably reversed this.

```
$ TZ=MST+7 date +%z
-0700
```

## Broken-down time

Unix represents calendar time using the following structure, which is consumed or emitted by several key functions:

```
struct tm
{
    int    tm_sec;   /* seconds [0,60] (60 for + leap second) */
    int    tm_min;   /* minutes [0,59] */
    int    tm_hour;  /* hour [0,23] */
    int    tm_mday;  /* day of month [1,31] */
    int    tm_mon ;  /* month of year [0,11] */
    int    tm_year;  /* years since 1900 */
    int    tm_wday;  /* day of week [0,6] (Sunday = 0) */
    int    tm_yday;  /* day of year [0,365] */
    int    tm_isdst; /* daylight saving flag */
};
```

Many of the most common problems with using the Unix time API can be traced to the rather confused and inadequate design of this structure. In what is probably unintentional humor, various manual pages and standards documents refer to it as "broken-down time".

The most notable pain points include:

- `tm_isdst` is easily misinterpreted as a flag when it's actually three-valued: 0 = DST off, 1 = DST on, -1 = DST status unknown. (Actually, values < -1 are usually treated as -1 and values > 1 treated as 1 - but do not count on this, as buggy implementations may treat values outside this range as offsets in hours.)
- The `tm_year` base value of 1900 rather than zero - easily forgotten and confusing, especially when interpreting negative values.
- Inconsistency about 0- vs. 1-origin in month and day numbers. The number of programmers who have gotten tm_mon wrong as a result is staggering.

and, worst of all,

- No timezone offset!

There is a potential overflow problem with `tm_year` on systems where sizeof(int) < sizeof(time_t); historically this was often true on 32-bit systems and may remain a hidden problem in embedded deployments. The year derived from a `time_t` can be so large that it won't fit in an `int`, which means `localtime(3)` and `gmtime(3)` could silently fail on valid time stamps in the far past or far future. A subtler problem is that when `tm_year` > INT_MAX - 1900 the Gregorian year doesn't fit in an `int`; a lot of code gets this wrong. Because INT_MAX on a 32-bit system is 2^31 = 2,147,483,647 this is unlikely to be a problem for normal historical dates.

Some versions of BSD remedied the last and most serious problem by adding two additional fields specifying the timezone used to generate the instance:

```
    long int tm_gmtoff;  /* time offset in seconds east of UTC/GMT */
    const char *tm_zone; /* name of timezone */
```

The GNU C library, used on most open-source Unixes, supports these members. The POSIX/SUS/OpenGroup standards allow them, but do not mandate them; in a strict ISO-C environment they will not be visible. You may have to `#define _BSD_SOURCE` before including `time.h` to make them visible.

The GNU C library documentation says this: "[The tm_gmtoff] field describes the time zone that was used to compute this broken-down time value, including any adjustment for daylight saving; it is the number of seconds that you must add to UTC to get local time. You can also think of this as the number of seconds east of UTC. For example, for U.S. Eastern Standard Time, the value is -5*60*60."

With the offset member (but not without it) a broken-down time specification is unambiguous.

The tm_zone member is less useful. The zone name strings it points to are not standardized and can be somewhat haphazard. They're not the IANA geographic designators. Under Linux and BSD they're typically the familiar (but ambiguous) three- and four-letter abbreviations (EDT, CEST, etc.), but under Windows they're longer strings like "W. Europe Standard Time". (And they're sometimes brain-bending monstrosities

like "GMT Daylight Time", which is Microsoft's own special name for what the Brits call British Summer Time. [MTZ])

## tzset(3) and ftime(3)

```
#include <time.h>

void tzset (void);

extern char *tzname[2]; /* time zone name */
extern long timezone;   /* seconds west of UTC, *not* DST-corrected */
extern int daylight;    /* nonzero if DST is ever in effect here */
```

The `tzset(3)` call computes the timezone offset for purposes of local-time computation. This information is expressed as three global variables (you may have to set feature macros to expose them).

These values are derived from system administrative settings and (if it is present) the value of the environment variable `TZ`, in ways too complex to delve into here; consult the manual pages for details. The key point is that after a `tzset(3)` call, the local timezone information should be available.

Note that the timezone offset does **not** include DST correction and is of opposite sign to the `tm_gmtoff` member.

The `tzname` variable is a pair of strings; `tzname[0]` is the zone name without and and `tzname[1]` is with DST correction. These will be civil zone names (like GMT/GST or EST/DST) not the IANA naming scheme.

`tzset(3)` is required by POSIX to be thread-safe, but access to the globals it sets is intrinsically thread-unsafe.

It should usually not be necessary to call `tzset(3)` directly, as it is called at the beginning of processing by most of the functions in this section. Exceptions will be noted below.

This part of the Unix time interface is very old; `tzset(3)` goes back to System III, and at least one of the globals it sets traces back to V6. Later portions of the Unix API design almost completely avoided predefined global variables in favor of returns from function calls.

Some early Unixes (including V7) had a a different way of fetching similar information; `ftime(3)`. This call filled in a structure with members containing information similar to the exposed globals above:

```
struct timeb {
   time_t          time;      /* seconds since epoch */
   unsigned short millitm;    /* milliseconds part of current time */
   short           timezone;  /* local time offset, minutes west of GMT */
   short           dstflag;   /* if nonzero, local zone uses DST */
};
```

The Single UNIX Specification carried this forward, but implementations were weak and defective. POSIX.1-2001/SUSv3 said that the contents of the `timezone` and `dstflag` fields are unspecified and should not be relied on. POSIX.1-2008/SUSv4 removed `ftime(3)` entirely. The archaic `tzset(3)` interface remains.

Final warning: some BSD Unixes do not implement `timezone` at all!

## gmtime(3) and localtime(3)

```
#include <time.h>

struct tm *gmtime(const time_t *);
struct tm *gmtime_r(const time_t *, struct tm *);
struct tm *localtime(const time_t *);
struct tm *localtime_r(const time_t *, struct tm *);
```

These are the basic functions for making a broken-down time structure `struct tm` from a `time_t`.

The difference between the gm* pair and the local* pair is this:

The local* pair computes the local time corresponding to the input Unix UTC time, taking the local time zone and all DST corrections into account. It does this by, in effect, adding the local timezone offset to the UTC seconds before performing the same computation performed by gmtime, although in a high-quality implementation it's more complicated than that, because of the possibility that the addition could overflow.

- Implementations that have a time zone history database, such as glibc with tzdata, apply the historical offset that was in effect at the time being converted according to the current revision of the IANA timezone history. This is normal on Unix systems. There is more detail on this behavior under the discussion of `mktime(3)`
- Implementations that lack such a database, such as Visual Studio runtime on Windows, apply the offset that would have been in effect at the time being converted according to the rules at the time the software shipped.

The gm* pair applies no such offset, so the output `struct tm` expresses Unix UTC seconds as UTC time.

The names of the gm* pair are a historical error, since they actually convert to UTC. These systems are not actually identical; for real GMT we would incorporate the DST bit, but for UTC it must be ignored.

The functions `gmtime(3)` and `localtime(3)` return a pointer to internal static storage that is overwritten on each call (in the GNU C library this static storage is shared among both functions). This is not thread-safe. The *_r functions are, on the other hand, re-entrant, with the user being required to pass in the address of the storage to be modified, and can thus be thread-safe.

POSIX implies that `localtime(3)` is required to behave as though `tzset(3)` has been called, but that `localtime_r(3)` is not (this is almost certainly because `tzset(3)` sets globals if localtime_r(3) also set them its callers could well become thread-unsafe".

For portable code `tzset(3)` should be called before `localtime_r(3)` - but note that this can lead to subtle errors when processing historical date/times because of the historical-date problem described under mktime(3).

Now that we've gone through the terchnical details, here is a less formal but more evocative way of summarizing them: localtime(3) is a pit of horrors that wants to be your personal hell - avoid. This is especially true if you are concerned about reproducibility (e.g. in unit and regression tests) where timezone- and DST-related problems can introduce random glitches like off-by-one-hour errors with no cause in your visible code.

## mktime(3), timelocal(3), timegm(3)

```
#include <time.h>

time_t mktime(struct tm *tm);
```

The `mktime(3)` function inverts `localtime(3)`, turning an input `struct tm` into a `time_t`. It is standardized.

There is no analogous standardized `mkgmtime()` function that inverts `gmtime(3)`, but see the description of `timegm(3)` below.

The `mktime(3)` function has a side effect that can trip you up seriously if you're not aware of it. In addition to returning a Unix seconds value, it also modifies the contents of the `struct tm`.

This means, in particular, that calling `mktime(3)` a second time on the same `struct tm` won't necessarily return the same value as it did on the first call! The most likely error is for the second return to be 3600 seconds off because the first call set the `tm_isdst` member.

The manual page says:

"The mktime() function modifies the fields of the tm structure as follows: tm_wday and tm_yday are set to values determined from the contents of the other fields; if structure members are outside their valid interval, they will be normalized (so that, for example, 40 October is changed into 9 November); tm_isdst is set

(regardless of its initial value) to a positive value or to 0, respectively, to indicate whether DST is or is not in effect at the specified time. Calling mktime() also sets the external variable tzname with information about the current timezone."

The last sentence is fairly subtle: it means that the implicit tzset(3) call at the beginning of mktime(3) can behave differently from an ordinary tzset(3) call, in that the implicit call can take advantage of knowing the time stamp that the caller is interested in, and can set global variables to values that are tailored for that time stamp. For example, if the time zone is Europe/London and the time stamp argument to mktime is circa 1970, mktime can set the global variables to values appropriate for a location that is at UTC+1 year round, which is what London was observing back in 1970; whereas if the time stamp is circa 2014, mktime can set the global variables to values appropriate for a location that is at UTC in winter and UTC+1 in summer, which is what London was doing in 2014.

The normalizing side-effect of mktime(3) makes it useful for various kinds of date/time arithmetic on broken-down time structures. For example, you can increment a tm_day member, call mktime(3) on it, and expect overflow beyond the end of month to be handled (but beware that this will be done without adjusting for leap-second discontinuities).

```
#include <time.h>

time_t timelocal(struct tm *tm);

time_t timegm(struct tm *tm);
```

These are GNU C Library extensions, not standardized. They are modeled on calls introduced in BSD and still present in FreeBSD/NetBSD/OpenBSD.

The `timelocal(3)` function is identical to standard `mktime(3)` (except for assuming `tm_isdst` is initially negative, which `mktime(3)` does not) inverting `localtime(3)`. The `timegm(3)` function inverts `gmtime(3)`.

The GNU C Library documentation recommends against using these functions, as they introduce a dependency. It recommends this as a portable alternative (but note that it is thread-unsafe):

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

time_t
my_timegm(struct tm *tm)
{
     time_t ret;
     char *tz;

     tz = getenv("TZ");
     if (tz)
         tz = strdup(tz);
     setenv("TZ", "", 1);
     tzset();
     ret = mktime(tm);
     if (tz) {
         setenv("TZ", tz, 1);
         free(tz);
     } else
         unsetenv("TZ");
     tzset();
     return ret;
}
```

## asctime(3) and ctime(3)

In older Unixes the following functions were available to report time and date as a string. They wired in bad design choices (the date string is of unpredictable length and includes a trailing "\n"), are not locale-aware, and have undefined behavior for years before 0 or after 9999.

```
#include <time.h>

char *asctime(const struct tm *);
char *asctime_r(const struct tm *, char *);
char *ctime(const time_t *);
char *ctime_r(const time_t *, char *);
```

These functions are obsolete and should not be used in production code; use `strftime(3)` instead. If you are curious about the details, read their manual pages.

## strftime(3)

```
#include <time.h>

size_t strftime(char *, size_t, const char *, const struct tm *);
```

This is the modern function for formatting timestamps into string representations; read its manual page. Here are some potentially interesting recipes that illustrate usage patterns:

```
/* RFC-3339 format */
strftime(buf, sizeof(buf), "%Y-%m-%dT%H:%M:%SZ", gmtime(t));

/* ISO-8601 local time */
strftime(buf, sizeof(buf), "%Y-%m-%dT%H:%M:%S", localtime(t));

/* RFC-822/RFC-2822 format */
strftime(buf, sizeof(buf), "%a, %d %b %Y %H:%M:%S %z", localtime(t));
```

RFC-822/RFC-2822 specifies English month and weekday names, but %a and %b are locale-aware; thus, the third recipe will deliver interestingly nonstandard results in a non-Anglophone locale.

Some of the format specifiers need to be used with caution; older implementations may not support all of them. Unit-test your formatting, checking carefully for %z and the other Single UNIX Specification and glibc additions (%C, %D, %E, %h, %n, %O, %P, %r, %t, %T, %V) if you must rely on them.

If called with a manually populated struct tm with all Standard C members populated, but without the `tm_gmtoff` and `tm_zone` members populated, strftime will, on some systems, not give sensible results for %z or %Z. Worse, it directly reads from the possibly-uninitialized `tm_zone` pointer.

More generally, %z and %Z for anything other than an immediate `localtime(3)`/`strftime(3)` pair can get wacky. In particular:

- If you're on a system without `tm_gmtoff` and `tm_zone`, they'll print the local time zone even for a gmtime(3)-initialized struct tm.
- If you change the value of TZ between `mktime(3)` and `strftime(3)`, if you have `tm_zone`, %Z will use a random zone for the new zone, not the abbreviation for the old one. If the new zone has fewer abbreviations than the old one, `tm_zone` (and thus %Z) can even end up pointing into freed memory!
- If you're constructing struct tm instances by hand before passing them to strftime (or, for that matter, mktime/gmtime), it's best to zero them out first, with memset or the equivalent.
- If you can help it, only call strftime on struct tm instances that localtime/gmtime have constructed for you. If you must construct them by hand, try to limit your strftime format specifiers to the "normal" ones that obviously correspond to fields you have set.

### strptime(3) and getdate(3)

```
#include <time.h>

char *strptime(const char *, const char *, struct tm *);
struct tm *getdate(const char *);
int getdate_r(const char *string, struct tm *res);
```

The `strptime(3)` function is a near-inverse of `strftime(3)` function, with one painful exception; it does not parse timezones. (If only because the misdesign of the standardized `struct tm` leaves it no place to put the information.) The full list of `strftime(3)` specifiers not supported are %F, %g, %G, %u, %v, %z, and %Z.

Do not use the %y specifier. Its interpretation differs incompatibly among implementations.

Test code that uses `strptime(3)` carefully on real data. Implementations are bug-prone.

The `getdate(3)` function and its re-entrant twin `getdate_r(3)` are complex and dubiously-designed attempts to do adaptive date parsing. We recommend against attempting to use them. See their manual pages for details.

## Berkeley multiple-timezone API

```
#include <time.h>

timezone_t tzalloc(const char *zone);
void tzfree(timezone_t tz);
const char *tzgetname(timezone_t tz, int isdst);
long tzgetgmtoff(timezone_t tz, int isdst);
char *ctime_rz(timezone_t tz, const time_t *clock, char *buf);
struct tm *localtime_rz(timezone_t tz, const time_t * clock, struct tm *
result);
time_t mktime_z(timezone_t tz, struct tm *restrict tm);
```

Some Berkeley Unixes have a group of functions designed to avoid any dependence on the system timezone globals, thus making it easier to handle multiple timezones in one program. This approach is also good for re-entrancy.

Because these functions are not standardized, this section is only a sketch intended to convey the style of the interface; for more details consult their manual pages.

`tzalloc()` takes a timezone specification and returns a `timezone_t` structure, which should later be freed by `tzfree()`. It is passed to *_rz functions which behave like the corresponding *_r functions, except that a hidden reference to system timezone globals is replaced by looking at a first argument which must be a pointer to a `timezone_t`.

This is how it should have been done in the first place...

## Good programming practice

As previously noted, localtime(3) is a pit of unreproducibility horrors due to the hidden and time-variable rules for computing timezone offsets. Avoid it.

To stay out of trouble, convert dates to Unix UTC on input, do all your calculations in that, and convert back to localtime as late as possible. This reduces your odds of introducing a misconversion and spurious timezone skew.

Any time you find yourself writing code that knows how many days there are in a month, something is probably very wrong. Reexamine your assumptions. You may want to bite the bullet and use `timegm(3)`, even

though it's nominally not portable.

Be very careful if you find yourself trying to do anything fancy with dates and times. If it's not obvious already, date and time processing is rather outrageously complex, and there's almost always one more quirk or gotcha lurking around the next bend that you haven't thought of yet. Whenever possible, offload the work to the standard library functions — their implementations have **most** of the bugs worked out by now.

A potentially useful reference on good practice in dealing with timezones and DST is [DSTBP]

## Improvements

The author welcomes corrections and improvements. This document has been written in the editorial *we* because he hopes to attract domain experts to cooperate on future revisions.

## Acknowledgments

Yuri Khan, Geoff Clare, Gary E. Miller, and Paul Eggert rendered substantial help with the beta revision of this document. Steve Summit wrote substantial critiques of versions 1.2 and 1.3.

## References


- [UTC] Wikipedia: Coordinated Universal Time
- [TIMESCALES] Time Scales
- [GMT] Wikipedia: Greenwich Mean Time
- [UNIX-TIME] Wikipedia: Unix time
- [POSIX-TIME] Rationale: Seconds since the Epoch
- [FP] What every computer programmer should know about floating point, part 1
- [TIME.H] Open Group: time.h
- [TIME-ZONE] Wikipedia: Time Zone
- [IANA-ZONES] List of tz database time zones
- [COMPUTERPHILE] The Problem with Time & Timezones
- [OLDTZ] Environment Variables
- [TZSOURCES] Sources for Time Zone and Daylight Saving Time Data
- [TZDATA] Wikipedia: tz database
- [ISO-8601] Wikipedia: ISO-8601
- [RFC-3339] RFC-3339
- [ Microsoft Time Zone Index Values]
- [DSTBP] Daylight saving time and time zone best practices


## Revision History

1.7: 2017-09-09
: Emphasize that localtime(3) is a horror to be avoided.

1.6: 2016-06-06
: Note difference between UTC and GMT. Note that NTP does not offer leap-second correction.

1.5: 2016-05-29
: Remove some buggy code pending better unit testing.

1.4: 2016-05-04
: Add information on Berkely multiple-timezone API.

1.3: 2016-04-24
: Various minor corrections and clarifications, mostly around timezones.

1.2: 2015-01-22
: Note that timezone is not necessarily supported by *BSD. Add a gotcha in the interpretation of the DST bit. More detail under mktime(3) on how historical dates are handled.

1.1: 2015-01-08
: Explain the contents of `tzname` better. New section on good programming practice.

1.0: 2014-10-09
: First production release.

0.9: 2014-09-29
: Beta version circulated for critique.

---

1. It has been alleged that "CUT" was avoided in part because it sounds like a rude word in Dutch

2. It is rumored that early versions of BeOS used float time, but the experiment was quickly abandoned.

3. In the mid-19th-century British clocks were sometimes made with two sets of hands, one for mean solar time and the other for railroad time.

4. The IETF is working on a protocol for pushing updates of the databases over the Internet.

5. The IANA database used to try to cover pre-1970 history as well, though some of this is being phased out since they have learned that their primary source for very old time zone history - a book on astrology - just made some stuff up.

---

Last updated 2017-09-09 11:35:11 EDT