Why Converting to and from UTC is Expensive and Unreliable

As Tom Scott has eloquently described, software to handle dates and times is more complicated than one might naïvely suppose. The overwhelming majority of the complications (and consequent bugs and code) come from the handling of at most about three hours per year, or one part in 2922 of the time. (You can double that in a few cases; and even rarer cases get another factor of four.) Those hours are the “transitions” of the zone, the nominal times at which clocks (or, in some rare cases, calendars) get adjusted forward or back.

Since the operating system itself has to deal with time, it does have information about those transitions and makes the information available to other software running under it. Naturally, the operating system's APIs are badly designed, making false naïve assumptions because their designers didn't know what they were dealing with. The subject of my talk today is just how bad that is.

There are two ways the O/S provides this information: one is the means to convert between local time and UTC; the other is information about general time zones, regardless of which the user has configured the system to use by default.

Microsoft's Time-Zone Data API

Let's see what you get. You pass an API one of these to fill in and a year for it to describe

```
typedef struct _TIME_ZONE_INFORMATION {
  LONG       Bias; // UTC - local time
  WCHAR      StandardName[32];
  SYSTEMTIME StandardDate; // End of DST, or invalid.
  LONG       StandardBias;
  WCHAR      DaylightName[32];
  SYSTEMTIME DaylightDate; // Start of DST, or invalid.
  LONG       DaylightBias;
} TIME_ZONE_INFORMATION, *PTIME_ZONE_INFORMATION, *LPTIME_ZONE_INFORMATION;
```

in which we see two

```
typedef struct _SYSTEMTIME {
  WORD wYear; // 1601–30827; but 0 in a transition rule
  WORD wMonth; // Jan = 1 through 12 = Dec; 0 marks struct invalid
  WORD wDayOfWeek; // Sun = 0 through 6 = Sat
  WORD wDay; // 1–31; or 1–5 for given day of week's repeat in month
  WORD wHour; // 0–24
  WORD wMinute; // 0–59
  WORD wSecond; // 0–59
  WORD wMilliseconds; // 0–999
} SYSTEMTIME, *PSYSTEMTIME, *LPSYSTEMTIME;
```

If either of those has `wMonth` set to zero, the other must have the same and the zone has no transitions (in some interval). Otherwise, both must be valid and they describe the year's transitions. This naïvely assumes a year either has no transitions or has two. That is not true.

- Zones do make changes to standard offset. Sometimes these replace a DST transition. This can lead to a year with a single transition.
- Some Islamic countries have interrupted DST for Ramadan, to make it easier for the faithful to stay up past sunset. (Thos fasting from sunrise to sunset have the opposite problem to the one “solved” by DST.)
- Some zones have experimented with “double DST”.
- A few zones have shifted the international date line.

For the common (first) case, MS sets one of the dates to the start or end of the year being described, to fake up having just one transition, (usually inaccurately) describing one side of it as DST and the other as standard. Comparing with data for adjacent years can give clues to the correct interpretation of this, but heuristics are untrustworthy.

There's a “dynamic” version of this, too, with all the same “limitations”. And MS doesn't bother much with historical accuracy, back past a few years before the release of whichever version of MS-Windows you're using. I'm fairly sure I've seen their data use USish transition dates for EU zones…

For local time, MS has its own versions of the POSIX `time_t` APIs, but I'll draw a veil over the differences between those and POSIX.

So much for The Evil Empire. The civilised world, in contrast, uses the IANA DB (formerly Olson database) either via ICU or by accessing the files under `/usr/{lib,share}/zoneinfo/`, which are compiled from the database into a compact (and fairly easy to digest) binary format. Now let's look at conversion between local time and UTC.

POSIX `time_t` APIs

Conversion between local time and UTC on Unix (and hence anything that implements relevant POSIX APIs) is taken care of by a bunch of functions involving `time_t` and `struct tm`. The former is a count of seconds after the UTC start of 1970. Officially, the latter's members are (in no particular order).

`tm_year`
: year number −1900

`tm_mon`
: month number, Jan = 0, …, 11 = Dec

`tm_mday`
: day of month, 1–31

`tm_hour`
: hour within the day, 0–23

`tm_min`
: minute within the hour, 0–59

`tm_sec`
: second within the minute, 0–60

`tm_isdst`
: 1 = DST, 0 = standard time, −1 = unknown

`tm_wday`
: day number within the week and

`tm_yday`
: day number within the year, 0 = Jan 1st

which `glibc` augments with:

`tm_gmtoff`
: Local time's offset from GMT at the moment described and

`tm_zone`

The zone abbreviation for local time.

and describe a moment in time, with respect either to UTC or to local time, depending on the context in which the structure isused. Whether the functions cope with date-times before 1970 is unspecified and MS's implementations don't; Darwin's do, but only back to 1900 (i.e. `tm_year` ≥ 0). All modern 64-bit platforms use a 64-bit `time_t` but some 32-bit systems still only use 32 bits and are duly headed for trouble in 2038. It's only 15 years away, now.

Some of the functions manipulating these return pointers to internal static buffers within `libc`; these typically have variants with an `_r` suffix on their names, used by prudent coders, that take a structure into which to write their results, instead of that static buffer. I shall ignore this, aside from noting that naturally you should call these instead of the one's that, for simplicity, I'll be discussing. The ones we'll be caring about here are:

`time_t mktime(struct tm *when)`
: The input is interpreted as local time; the output is a time-point it could represent. There's no `_r` version (no need). Notice it's parameter is not `const`-qualified: the `struct tm` it points to gets updated by this call. The fields of that struct listed above after `tm_isdst` are ignored as input and set suitably in the output. The fields listed above before `tm_isdst` need not lie in their standard ranges: POSIX specifies this function to accept a *denomal* representation of time and reduce it to normal form (each field within the given ranges). The return value on error is −1; but that's also the return value for the last second of 1969. This is not the only ambiguity.

`struct tm *localtime(cont time_t *tp)`
: The output gives local time's description of the input. This is unambiguous.

`struct tm *gmtime(const time_t *tp)`
: The input gives UTC's description of the input. Note that this is just “trivial” arithmetic (albeit including some calendarical calculations).

I have no complaints about the last two, albeit I might show you some code that used them … clumsily. All the fun is with the first.

The problem with `mktime()` is, inevitably, transitions. If its input describes 02:30 on a recent Sunday (the last one in March) and your local time is CET, it is technically invalid: that Sunday had no 02:30, it skipped from just before 02:00 to 03:00. Nothing I've spotted in `mktime()`'s spec precludes it treating this as error, which would be awkward. Fortunately, every implementaiton I know of (even MS's)

does “something sensible” instead; it picks a time before or after the transition that you might plausibly have meant. It gives you no clue that it's done that, though, other than changing the details in the input `struct` from what you set.

If the input describes 02:30 on the last Sunday in October, still in CET, there are two interpretations: `mktime()` will pick one of them for you. Maybe it's the one you needed, maybe you needed the other. It won't tell you there was another you might have wanted to know about.

The above is for when you've set `tm_isdst` to −1. If you set it to 1 or 0, declaring that you know whether the input describes a time in DST or not, the ambiguous case will give you the one you asked for. The technically invalid case, on the other hand, will flip your `tm_isdst` setting to select the time the input would describe, were its `tm_isdst` in effect, re-expressed to take account of the fact that you set it wrong. This is actually sensible and part of the “cope with denormal” behaviour – it's also what it does for normal times, if you specify `tm_isdst` wrong. That actually provides a way to discover the standard time offset at a specified time, something that ECMA 262 requires for the implementation of `Date`:

```
    tzset();
    const time_t curr = time(nullptr);
    if (curr != -1) {
        struct tm t;
        if (gmtime_r(&curr, &t))
            return int(curr - mktime(&t));
    } // else, presumably: errno == EOVERFLOW
```

which replaced an older

```
    struct tm t;
    time_t curr;
    tzset();
    time(&curr);
    localtime_r(&curr, &t);
    time_t locl = mktime(&t);
```

```
    gmtime_r(&curr, &t);
    time_t globl = mktime(&t);
    return (double(locl) - double(globl)) * 1000.0;
```

whose flaws are so many and varied that one of them was that working out what it actually did – and how that managed to approximate correctness – actually impeded my ability to think about the problem coherently. Kludges rot the mind of the reader.

I wish `mktime()` accepted an optional second parameter of the same type. Because when the input is ambiguous, that would let it fill in the second with the other candidate's data. This would make life so much easier.

Edward Welbourne