SSOReady

Back to Blog

Australia/Lord_Howe is the weirdest timezone

Timezones are weird. But only finitely so. Here's the exact conceptual model you should have of them.

Ulysse Carion
Cofounder and CTO, SSOReady

The standard trope when talking about timezones is to rattle off falsehoods programmers believe about them. These lists are only somewhat enlightening – it's really hard to figure out what truth is just from the contours of falsehood.

So here's an alternative approach. I'm gonna show you some weird timezones. In fact, the *weirdest* timezones. They're each about as weird as timezones are allowed to get in some way.

- `` `Asia/Kathmandu` `` has a weird offset from UTC
- `` `Africa/Casablanca` `` doesn't fit into the timezone model cleanly, so it's hard-coded
- `` `America/Nuuk` `` does daylight savings at -01:00 (yes, with a negative)
    - and `` `Africa/Cairo` `` and `` `America/Santiago` `` do it at 24 o'clock (not 0 o'clock)
- `` `Australia/Lord_Howe` ``, population 382 and some notable stick bugs, has the weirdest daylight savings rule

To learn how their weirdness is represented in software, we'll look at the raw timezone files that all software ultimately relies on. From there, two things will become clear:

- Yeah, this stuff is weird
- But only finitely so, because ultimately a computer's gotta implement them

But first, an aside on the calendar.

PGXIIREAM: Pope Gregory XIII rules everything around me

Unless you're doing some fairly exotic things where you're finding yourself saying things like

> *"Oh yeah the OCR on Japanese driving licenses pops out things like "□□ 8", that's just how they sometimes say 1996 over there. That's why we have this in the parser:*
>
> ```
> eras = { "□□": 1912, "□□": 1926, "□□": 1989 }
> ```
>
> *One of these days we'll need to add `"□□": 2019`, but it hasn't come up yet."*

or

> *"We're gonna need to set up a per-country feature flag when deciding whether banks are closed for Eid. Saudi Arabia and Iran don't agree on when the lunar month starts."*

Then yeah, sure, you may need to write software that knows about the Japanese or Islamic calendar systems.

Cases like this are a small minority. The reality of the world is that the Western system of timekeeping is the dominant one, and even in e.g. Japan and the Muslim world, almost everyone who uses computers is familiar with the Gregorian system.

With computers, we project the Gregorian system into the future and past, which is called the proleptic Gregorian calendar and isn't historically accurate but nobody really cares except Russian revolution nerds.

This calendar system is pretty much good enough, and barring any rationalist coups d'etat, is the one we'll be stuck with for a long time. It does one thing well: it's very good at keeping the sun at the same place in the sky across the years. It doesn't let the months drift around the seasons like the Roman calendar did.

Technically, this “keep the sun roughly in the same place whenever it’s the same time-of-day” is called “mean solar time”. And that’s why GMT, Greenwich Mean Time, is called that way. It’s about the mean solar time of the English observatory in Greenwich.

By the way, we technically don’t call it GMT anymore. Unless you’re talking about what time people in London say it is, you probably technically mean UTC.

Coordinated Universal Time is basically just a modern formalization of GMT. It’s useful because almost everyone on the planet has agreed to base their clocks off of an *offset* from UTC. It’s still a solar mean time, but the connection to Greenwich isn’t really there anymore.

I bring this up because you may have heard of a weird modern quirk on Pope Gregory’s sun-following endeavors:

Leap seconds don’t matter

The Earth’s rotation is slowing down. Days are getting longer. So you need to correct for it if you want to keep IRL days in sync with computer days.

The nerd task force assigned to this problem is the International Earth Rotation and Reference Systems Service, which has two primary goals:

1. Watch the Earth rotate, and report back on their findings

2. Break Wikipedia’s CSS with their long name

timecops

If the days are getting longer, and they’re doing so at a fairly unpredictable rate, the simplest solution is to have IERS occasionally just insert an extra second in the day to make clocks go slower. It’s called a leap second.

You should completely ignore the fact that this is a thing. It’s a cool novelty, but it’s effectively just a detail you can ignore, because:

1. It’s not like programming languages support representing 61-second minutes anyway
2. You (and by you I mean your cloud provider) can just run your clocks slower around the time of the leap second, and pretend to everyone else over NTP that their clocks are running fast. This is called leap smearing.

Btw it’s called UTC (Universal Time Coordinated? huh?) because the same folks who publish UTC also publish UT1, which is UTC sans the leap seconds. There were other UTs before the Coordinated variant came up.

Weird time zones

OK! Let’s start looking at some weird time zones, and find out how your computer knows to represent them.

`Asia/Kathmandu` is on a weird offset

Most of the world is on a whole number of hours before or after UTC. About a fifth the world by population is on a half-hour offset from UTC; in particular, India is 5h30m ahead of UTC.

Nepal is 5h45m ahead of UTC:

```
$ TZ=UTC date ; TZ=Asia/Kathmandu date
Tue Jul 30 23:52:11 UTC 2024
Wed Jul 31 05:37:11 +0545 2024
```

If you’re like me, you must be have at one point wondered how in the *world* your computer knows this fact.

Here’s a hint:

```
$ TZ=Asia/Kathmandu strace -e trace=openat date
...
openat(AT_FDCWD, "/usr/share/zoneinfo/Asia/Kathmandu", O_RDONLY|O_CLOEXEC) = 3
Wed Jul 31 05:40:49 +0545 2024
```

On your filesystem is a database called the IANA Timezone Database, aka tzdb or zoneinfo. It’s a bunch of binary files, encoded in Timezone Information Format. The names of those files act as timezone identifiers, which is where you see strings like **`America/Los_Angeles`** or **`Europe/London`** come from:

```
$ tree /usr/share/zoneinfo
...
├── America
│   ├── Los_Angeles
├── Europe
│   ├── London
...
```

At the very end of `/usr/share/zoneinfo/Asia/Kathmandu` is this little string:

```
cat /usr/share/zoneinfo/Asia/Kathmandu
...
<+0545>-5:45
```

The syntax is here pretty obtuse, but what it means is:

> *“Unless otherwise specified, UTC is 5h45m behind this timezone. Call this time `+0545`.”*

That’s precisely how software can figure out the time in Nepal. That’s also why the output from `date` above has `+0545` in it.

Why strings like `PDT` or `CET` are pretty meaningless

In the example above, `+0545` is called a “designator”. It’s a pretty-ish string describing which *part* of a timezone a timestamp is in. It’s meant to be used for outputting timestamps, and is only unambiguous if you already know what timezone the timestamp was taken in.

Just *how* ambiguous are these designators? I wrote a `tzdump` script that converts TZIF files to JSON. Here’s the top hits:

```
find -L /usr/share/zoneinfo -type f \
  | xargs -n1 ./tzdump \
  | jq -r '"\(.ID)\t\(.Transitions[].LocalTimeType.Designation)"' \
  | sort | uniq | sort -k 2 | uniq -f 1 -c | sort -n | awk '{ print $1 "\t" $3
```

The most popular designators are:

```
66  CST
58  CDT
56  CET
56  CEST
```

A total of **66 timezones** use `CST`, either in the past or future. Many timezones are functionally exact clones of each other – there's no difference between `America/Phoenix` and `America/Creston`, but they each get their own file – but still. There's a lot of ambiguity in there.

In case you're curious, only 33 designators are unique to a timezone. A lot more are functionally unique, but I'm too lazy to dedupe logically-equivalent timezones right now.

As an extra fun bit of trivia, designators are not strictly uppercase/numeric. `ChST`, appearing in `Pacific/Saipan`, stands for Chamorro Standard Time. It's the only designator with a lowercase name. `CHST` is not taken, sadly for those of us who love bugs.

How are timezones with DST represented?

When we looked at Kathmandu, we had this string telling us the Nepalese time rules:

```
<+0545>-5:45
```

Ok, simple enough. But what about a timezone with DST transitions? The syntax has lots of defaults (DST will be a one-hour jump, it happens at 2am by default, etc) but `Europe/Athens` is a good example of one that uses most of the syntax:

```
$ cat /usr/share/zoneinfo/Europe/Athens
...
EET-2EEST,M3.5.0/3,M10.5.0/4
```

That syntax means:

> *"Standard time is called `EET`, it's 2 hours ahead of UTC. DST is called `EEST` (it's 3 hours ahead, an implicit default relative to standard time). Start DST in month `3` on the last instance of (`5`) day `0` (Sunday) in that month, at 3am local (`/3`). End DST on month `10` on the last Sunday at 4am local (`5.0/4`)."*

So yeah, your computer does a bunch of kind of gnarly logic to figure out what date-and-time a timestamp corresponds to, then figures out whether it's inside or outside DST to figure out the current local time. Delightful.

In case you're curious, the spec says "5" means "last instance of", and "1" means "first instance of". But only weeks "1", "2", and "5" are used:

```
$ find -L /usr/share/zoneinfo -type f | xargs -n1 ./tzdump | jq -r 'if .Rules.
18  1
89  2
81  5
```

Here's a fun twist: on my Mac 100% of timezones either don't have DST at all or use this nth-instance-of-day-of-month rules to do DST switching. But inside **`/var/db/timezone`** there's different versions of tzdb. In there is a version with other kinds of timezones in it:

```
$ cat /var/db/timezone/tz/2024a.1.0/zoneinfo/Africa/Casablanca
...
XXX-2<+01>-1,0/0,J365/23
```

That timezone basically means "we are perpetually on daylight savings", because the **`J###`** syntax means "`###`-th day of the year, not counting Feb 29 if there is one" (J stands for "Julian calendar").

Technically, that timezone also exercises the prefixless (i.e. without **`M`** or **`J`**) syntax for indicating days, where **`###`** means "`###`-th day of year, counting any Feb 29". But in this case it's a distinction without a difference.

(Aside: All this stuff comes from POSIX. GNU's docs about the POSIX **`TZ`** env var, which TZIF builds on, are the best I know of online for this stuff.)

But this is just the start of the weirdness that is **`Africa/Casablanca`**.

`Africa/Casablanca` and `Asia/Gaza` follow the moon, but timezones follow the sun

The TZIF format supports three possible rules for deciding on your daylight savings transition day:

- Rules like “first Tuesday of March”
- Rules like “45th day of the year”
- Rules like “45th day of the year, Feb 29 doesn’t count”

Morocco and Gaza do their daylight savings based on Ramadan. Ramadan is a month in the Islamic calendar. The Islamic calendar is based on the moon. The lunar calendar isn’t a clean multiple of the solar calendar; from the Gregorian perspective, lunar months seem to slowly “rotate” around the year, because they’re basically on a different modulo. There’s a problem there for our heroes at the tzdb.

The solution? The dumbest possible one.

A TZIF file ends with the footer syntax we’ve been talking about to this point. But it *starts* with a big long list of historical data about a timezone. If a country ever changes timezone rules, TZIF represents that by encoding the new rule in the footer, and hard-coding all the old transitions.

But you can also just take these hard-coded transitions and put them into the future. The hard-coded transitions take precedence over the footer. So the TZIF folks:

1. Picked a year far enough into the future (2086, as it turns out)
2. Wrote a script in emacs lisp to calculate Ramadan
3. Use the output of that script to generate transitions for Morocco and Gaza

And that’s why in practice Morocco and Gaza are just hard-coded in the tzdb, unlike every other timezone.

In case you’re hoping for more fun timezones like this, I’m afraid you’re out of luck. The others at the bottom of this list, which filters for transitions beyond 2025, are just synonyms of Casablanca and Gaza.

```
$ find -L /var/db/timezone/tz/2024a.1.0/zoneinfo/ -type f | xargs -n1 ./tzdump
  26 /var/db/timezone/tz/2024a.1.0/zoneinfo//Africa/Cairo
...
```

```
    26 /var/db/timezone/tz/2024a.1.0/zoneinfo//US/Pacific
    26 /var/db/timezone/tz/2024a.1.0/zoneinfo//WET
    26 /var/db/timezone/tz/2024a.1.0/zoneinfo//posixrules
   130 /var/db/timezone/tz/2024a.1.0/zoneinfo//Africa/Casablanca
   130 /var/db/timezone/tz/2024a.1.0/zoneinfo//Africa/El_Aaiun
   184 /var/db/timezone/tz/2024a.1.0/zoneinfo//Asia/Gaza
   184 /var/db/timezone/tz/2024a.1.0/zoneinfo//Asia/Hebron
```

It looks like every other timezone just has 26 transitions beyond 2025, which I think are just there to make software that doesn't know about the TZIF footer transition rules be accurate a few years into the future anyway.

`America/Nuuk` transitions to DST at -1 o'clock

Nuuk is in Greenland, and is part of the greater EU cinematic universe.

All of Europe (idk whether this is an EU/EEZ/EFTA/CoE thing) syncs up their daylight savings, except for Iceland, which doesn't do DST at all (`Atlantic/Reykjavik`, which is technically an alias for `Africa/Abidjan`, is basically just UTC; their rule string is just `GMT0`).

Most Europeans are familiar with three major timezones, which we can refer to as `Europe/Lisbon` (western), `Europe/Brussels` (central), and `Europe/Athens` (eastern). They're each one hour ahead of the other, and so their timezone transitions look like:

```
# I'm gonna space these out to highlight the symmetry,
# and also spell out the implicit "/2"

Europe/Lisbon:   WET0WEST ,M3.5.0/1,M10.5.0/2
Europe/Brussels: CET-1CEST,M3.5.0/2,M10.5.0/3
Europe/Athens:   EET-2EEST,M3.5.0/3,M10.5.0/4
```

In other words, Lisbon springs forward at 1am, Brussels at 2am, and Athens at 3am. But those times are *local*. In reality, they're all at the same instant.

This makes good sense. It's good for business that the time difference between any two spots in Europe is always the same.

Greenland would like to be part of the action. Thing is, Greenland is pretty far west of continental Europe. Whereas Lisbon's standard time is UTC, Greenland's is 3 hours behind UTC. Here's their daylight transition rules:

```
$ cat /var/db/timezone/tz/2024a.1.0/zoneinfo/America/Nuuk
<-02>2<-01>,M3.5.0/-1,M10.5.0/0
```

Take note of `M3.5.0/-1`. The first part is the standard European DST start day. The `/-1` part? That means that instead of doing DST at like 2am (`/2`), Greenland does it at -1 o'clock (`/-1`). The way the rules file is encoded, daylight savings for Greenland is meant to happen on Sunday, but in fact happens at 11pm on the Saturday before. Super weird.

I'm guessing this breaks software, because America/Nuuk and its aliases are one of those timezones whose transition rules are just entirely ommitted in `/usr/share/zoneinfo` on my Mac. They're only available in other copies of tzdb in `/var/db/timezone`.

Oh, `America/Santiago` and `Africa/Cairo` transition at 24 o'clock

Nuuk is the earliest anyone does a transition. Santiago and Cairo are the latest. They both do transitions at 24 o'clock? Like, the next day?

```
America/Santiago: <-04>4<-03>,M9.1.6/24,M4.1.6/24
```

```
Africa/Cairo: EET-2EEST,M4.5.5/0,M10.5.4/24
```

I think they're both encoded like that because of weirdness in how the governments define the rules. Like `M10.5.4/24` means "last Thurday of October, 24 o'clock", which really means "the day after the last Thursday of October". But that's not the same thing as "last Friday of October" if the month ends on Thursday?

Both of these files are also in Mac's list of naughty timezones that don't go in `/usr/share/zoneinfo`.

`Australia/Lord_Howe` has the weirdest DST transition

When you do a DST transition, you “spring forward” and “fall back”. *Surely* everyone agrees it’s a *one-hour* jump, right?

Here’s a script to check. What is the time difference between standard and daylight time in every timezone?

```
$ find -L /usr/share/zoneinfo -type f | xargs -n1 ./tzdump | jq 'if .Rules.DST

410 0
2   1800
185 3600
1   7200
```

Hmm. 410 timezones just don’t DST at all. 185 have a 3600-second, i.e. 1-hour, difference. And then there are the malcontents.

The 7200-second, i.e. 2-hour, jump is `Antarctica/Troll`. Fitting.

```
<+00>0<+02>-2,M3.5.0/1,M10.5.0/3
```

So during the winter (i.e. the northern summer) they use Norway time? But there are like 6 people over the winter at Troll? Do these 6 souls appreciate their contribution to software esoterica? I hope they do. Apparently they use like four different times during the year down there in practice, but there’s no syntax to express that.

OK but the real question is what’s up with the two 1800 transitions. They’re synonyms for each other. It’s `Australia/Lord_Howe`, which has a **powerful** 30-minute DST transition:

```
<+1030>-10:30<+11>-11,M10.1.0,M4.1.0
```

10h30m ahead of UTC standard, 11h DST. Love this for them. Running cron jobs on an hourly basis doesn’t in practice have very weird interactions with DST.

Everywhere else on the planet, every 60 minutes you're back to the same spot on the clock.

Except Lord Howe Island. Heroes. On the first Sunday of October, a 60-minute timegap only puts you halfway around the clock. All your cron jobs are now staggered relative to the local wall clock.

In case you're curious, Lord Howe Island belongs to Australia. It has 382 people at the latest census. It's a bit of a natural paradise, and apparently to preserve that there's a cap of 400 tourists at a time.

Probably the most famous aspect of Lord Howe is Ball's Pyramid.

Ball's Pyramid Memorial for Stickbugs and Software Engineers who write Timezone-related Code.

It's an old collapsed volcano. It looks cool. It has some rare stick bugs.

Big takeaways

Timezones are weird, but finitely so. All they consist of is:

- An ID, e.g. `` `America/Los_Angeles` ``
- A set of hard-coded transitions, which range from the past into the future
- A set of rules for how future transitions may happen

Any given time in a timezone is just:

- An offset from UTC
- With a “designator” time that doesn’t mean much
- (This usually isn’t outputted anywhere) Whether the time is considered DST

You can always uniquely identify what UTC time someone is referring to whenever they tell you their timezone + local time + current time designator. The timezone + designator gives you an offset, and you can apply the offset to the local time to get UTC.

Like, it’s weird, it’s quirky, but it’s not like all bets are off.

Also:

- Don’t let people bully you into thinking that just because something is complicated, it’s impossible.
- This is because almost every standard (except ISO8601, whatever) is just a file, and you can read it. You are smart. You can do it. Embrace the weirdness of Greenland’s daylight savings. Believe in yourself.
- If I were UN secretary general, I would kick out any countries that I deem insufficiently considerate of Paul Eggert’s time.

Appendix: Other weird stuff in zoneinfo

Honestly, there’s some stuff in zoneinfo that I can’t figure out. Even I have nerd-sniping limits. Exercises for the reader.

These time zones have *hundreds* of hard-coded transitions out into the future. I don’t understand why, it’s not like they all have lunar calendar stuff going on.

- Asia/Jerusalem has 780 transitions in the future, out of 901 total
- Africa/Cairo has 800 transitions in the future, out of 929 total
- America/Nuuk has 800 transitions in the future, out of 889 total
- America/Santiago has 800 transitions in the future, out of 931 total
- Pacific/Easter has 800 transitions in the future, out of 911 total
- Asia/Gaza has 982 transitions in the future, out of 1106 total

They all lack a rules footer, but our friend Africa/Casablanca has a mere 132 transitions hard-coded and lacks a rules footer too. What’s up with that?

P.S. If you’re the type of weird to think this stuff is neat: email me. ulysse.carion@ssoready.com ;)