U.S. DEPARTMENT OF ENERGY

PNNL-22604

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

Ordinal Date Library

DES-0066

Revision 1

Charlie Hubbard

May 2013

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by **Battelle** *Since 1965*

Ordinal Date Library

Charlie Hubbard

DES-0066
Revision 1
May 2013

Contents

1 Introduction

On Linux and other Unix-like (POSIX) operating systems, it is often most convenient to store absolute time stamps as `time_t` values (see below). In the project code base, `time_t` time stamps are used everywhere. However, at least one customer prefers to use time stamps formatted as *ordinal dates* in reports, data files and some file names. To ease conversions between these two time stamp formats, we have created the Ordinal Date library.

The Ordinal Date library is defined and implemented by the two files *ordinalDate.h* and *ordinalDate.cpp*. The primary documentation for the library is its source code and its associated Doxygen-generated HTML files. The Doxygen documentation should be consulted for a detailed description of the library's API. The document you are reading now is supplemental, and is intended to provide deeper background for the library's implementation. In cases where the Doxygen documentation disagrees with this document, the Doxygen documentation should be considered correct.

1.1 `time_t` Time Stamps

`time_t` values are 32-bit signed integer values that simply contain the number of seconds that have elapsed since midnight on the moring of January 1st, 1970 UTC[1]. This date is sometimes referred to as the *Unix Epoch*. It corresponds to a `time_t` value of zero. Dates before the Unix Epoch can be handled by negative `time_t` values.

1.2 Ordinal Date Time Stamps

Ordinal dates, at least in the context of this library, consist of at least a 4-digit *year* field indicating the year and a 3-digit *year-day* field indicating the number of days since the beginning of the year (where '1' means January 1st). Note that specific calendar dates do not always correspond to the same specific year-day number because the year-day number is affected by leap years. As an example, May 29th, 2012 (a leap year) matches the ordinal date

2012-150

but the same calendar day (May 29th) in 2013 (*not* a leap year) matches the ordinal date

2013-149

instead.

Note that the above two ordinal date time stamps only have day resolution, but `time_t` time stamps have second resolution. If higher resolution is required in an ordinal date, it is implemented as a decimal fraction attached to the year-day. For example, 6:00am represents

[1] *Coordinated Universal Time*, or *UTC*, is a closely related successor to *Greenwich Mean Time* (GMT). For the purposes of this document, the two can be considered synonymous.

a time that is one quarter through the day, so it is encoded by appending 0.25 to the year-day. So, for May 29th 2013, that looks like this

2013-149.25

Obviously the decimal fraction can be extended to as many digits as needed to achieve the necessary time resolution. For instance, 11:24:36am On May 29th, 2013 encodes as

2013-149.47542

with 1 second resolution.

1.3 Local Time vs. UTC

`time_t` time stamps are always relative to UTC because the Unix Epoch is based on UTC. Ordinal dates, however, can be with respect to UTC or to any other time zone (most typically to the local time zone). This needs to be considered when converting from one format to the other. The library handles this with a *local time* flag, which, when set to `true`, treats the ordinal date as if it were relative to the local time zone. Otherwise, it is assumed to be relative to UTC.

2 Implementation

The library implements four functions, the prototypes of which are shown below.

```
time_t OrdinalDateToUnixTime(const int year,
                             const int yday,
                             const int hour,
                             const int min,
                             const int sec,
                             const bool localTime = false);

time_t OrdinalDateToUnixTime(const int year, const double yday, const bool localTime = false);

void   UnixTimeToOrdinalDate(const time_t t,
                             int &year,
                             int &yday,
                             int &hour,
                             int &min,
                             int &sec,
                             const bool localTime = false);

void   UnixTimeToOrdinalDate(time_t const t, int &year, double &yday, const bool localTime = false);
```

The first two functions convert an ordinal date value to a `time_t` value. The second two perform the opposite function. Note that there are versions of each function that expect the ordinal date to specify the time portion explicity as *hours*, *minutes* and *seconds*, and versions that expect the time component to be expressed as a decimal fraction appended to the year-day value.

As previously mentioned, all functions take a boolean flag called `localTime`, which, when set to true means the ordinal date will be considered to be relative to the local time zone. Otherwise, they are taken as relative to UTC. UTC is the default for all of the functions, so the parameter doesn't need to be specified at all if UTC is the desired format.

2.1 `struct tm` Normalization

For the most part, the conversion functions in the Ordinal Date library are completely straightforward; but there are a couple of tricks that probably deserve a closer look. The first of these is `struct tm` normalization.

Internally, the Ordinal Date library functions rely heavily on the Unix `struct tm` data structure (defined in the standard header file */usr/include/time.h*), a copy of which is shown below.

```
struct tm {
   int tm_sec;   // seconds [0,61]
   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 savings flag
};
```

In the above code block, the comments specify the ostensibly valid range for each field, but it turns out that the system time functions that take a `tm` struct as input (including `mktime()` and `timegm()`, which we use internally) do not expect these ranges to be strictly adhered to. Instead, these functions take the structure field values however they are provided by the user, and then *normalize* them such that, afterward, the structure's fields have been adjusted to move them all within their legal range. As an example, if you set up the structure to say it is the 27th hour of July 1st, the structure will automatically be normalized such that afterward it properly indicates this same time as the 3rd hour of July 2nd.

We exploit this behavior for easy conversion of ordinal dates. The trick is to initialize the `tm` struct such that the month (`tm_mon`) is always set to January (0), and the day of the month (`tm_mday`) is always set to the user-provided year-day. From our "May 29th" example above, we'd initially set up the structure so that it was saying that it was January 149th, 2013. Then `timegm()` (or `mktime()`) normalizes the `struct tm` fields so afterward the appropriate fields indicate year=2013, month=4 (May), and day-of-month=29. This normal form is then converted to to a `time_t`.

This is great! Because the system library time functions handle this normalization of the `tm` structure automatically, we don't have to compute month and day-of-month values from the year-day value ourselves *and*, as a part of the normalization, leap years are also automatically accounted for, so we don't have to worry about that either.

2.2 Daylight Saving Time

The only other tricky part of the Ordinal Date library (to the extent that there is anything tricky to it at all) has to do with Daylight Saving Time (DST). When the `localTime` flag is set to `false` (the default condition), all conversions are to/from UTC. There are no seasonal changes for UTC, so daylight saving time doesn't come into play. However, dealing with local times, the code may or may not have to adjust for DST depending on whether or not it is in effect for the local time zone for the ordinal date specified (or being returned).

Fortunately, the `struct tm` structure has a field called `tm_isdst` that we can use to tell the system time functions how to proceed. `tm_isdst` can be set to one of three values – 0, 1 or -1. These are interpreted as follows by the system time functions.

- **`0`** — Daylight Saving Time *is not* in effect
- **`1`** — Daylight saving Time *is* in effect.
- **`-1`** — Let the system try to decide whether or not DST is in effect for the local time zone based on the date configured in the `tm` structure.

For conversions to/from UTC, DST is never in effect so we always set `tm_isdst` to zero in that case. For local time conversions, we first set this field to -1 before calling the appropriate system time function, and the system function automatically adjusts for DST as necessary.

Unfortunately, the system can not make this determination completely unambiguously. In particular, consider the case when we're switching from DST back to standard time. At the appointed hour (typically 2:00am on a Sunday morning), we set our clocks back an hour. That is, it was 2:00am, and then, in an instant, it is 1:00am again. On that night, clocks actually go through the time interval between 1:00am and 2:00am twice – once under DST and once again under standard time. So, for instance, if you were to configure the `tm` struct for 1:37am on that day, the system time library has no way of knowing if you are referring to the first 1:00am to 2:00am interval (DST) or the second one (standard time). The results are unpredictable and may be different on different operating systems or different versions of the same operating system. The reverse situation, switching from standard time to DST, creates a similar, albeit somewhat less dire, situation.

The Ordinal Date library makes no attempt to fix this problem. For that reason, unless there is a compelling reason to do otherwise, users should stick to UTC-only conversions (this is exactly the reason the library functions' `localTime` flag defaults to `false`), or at least be aware that there can be unexpected behavior at the switch from/to DST.