[contents]

Working with Time Zones

W3C Working Group Note 5 July 2011

This version:
http://www.w3.org/TR/2011/NOTE-timezone-20110705/
Latest version:
http://www.w3.org/TR/timezone/
Previous version:
http://www.w3.org/TR/2005/NOTE-timezone-20051013/
Editor:
Addison Phillips, Invited Expert
Additional Contributors:
Norbert Lindenberg, Yahoo!
Mark Davis, Google; Unicode Consortium
Martin J. Dürst, Aoyama Gakuin University; W3C Internationalization Interest Group Chair
Felix Sasaki, DFKI GmbH
Richard Ishida, W3C

Abstract

This document contains guidelines and best practices for working with time and time zones in applications and document formats. Use cases are provided to help choose an approach that ensures that geographically distributed applications work well with date and time values. This document also aims to provide a basic understanding and vocabulary for talking about time and time handling in software, a source of confusion for many developers and content authors on the Web.

Status of this Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

This document contains best practices and user scenarios for working with time zones. Date and time values can be complex and the relationship between computer and human

timekeeping systems can lead to problems. This document aims to provide a basic understanding and vocabulary for talking about time and time handling in software. The working group has updated this document to contain more comprehensive guidelines and best practices for working with time and time zones in applications and document formats. Use cases are provided to help choose an approach that ensures that geographically distributed applications work well.

This document is a W3C Working Group Note. It has been produced by the i18n Core Working Group, which is part of the Internationalization Activity.

Please send comments on this document to www-international@w3.org (publicly archived).

Publication as a Working Group Note does not imply endorsement by the W3C Membership. This document may be updated, replaced or obsoleted by other documents at any time. Therefore, quotes or references to specific information in the document should include the publication date of this version, 05 July 2011. It is inappropriate to cite this document as other than a Working Group Note, which is not an endorsed W3C Recommendation.

This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

Table of Contents

1 Introduction

One common requirement for applications is the need to deal with dates, times, or durations. Working with time-related data can be complex because values are related to calendars and timekeeping rules, which themselves may be somewhat arcane. One of these complexities in working with time-related data is the effect of time zone on the data.

This document contains guidelines and best practices for working with time and time zones in applications and document formats. Use cases are provided to help choose an approach that ensures that geographically distributed applications work well with date and time values. This document also aims to provide a basic understanding and vocabulary for talking about time, a source of confusion for many developers and content authors on the Web.

Some W3C documents related to these guidelines include:

[HTML5] defines a variety of date and time string types that can be used for input and display.

[XML Schema] provides a variety of data types for dates and times, such as `date`, `time`, and `dateTime`. These data types follow internationally friendly formats defined by ISO 8601 and can be used to address a variety of differing date or time applications.

The various XML Schema time types can either include or omit the offset from Universal Coordinated Time (UTC). The presence (or absence) of this offset implies different things about the time value represented and presents different processing issues for applications. The same is true of HTML5's "local date and time" or "global date and time" types.

Note: Users and implementers of languages or specifications which handle time-related data should take the following recommendations into account even if time-zone-sensitive data is rarely used. Sooner or later some data will be affected by the issues described. Some examples of these include XQuery, XPath, and XSLT.

2 A Brief History of Timekeeping

Computer systems tell time differently than people do. So it is helpful to understand how time works within computers as well as in the real world in order to get a handle on the things that can go wrong.

2.1 Observed Time

Most timekeeping systems are organized around observable events, typically celestial ones such as sunrise, sunset, longest and shortest day of the year, the sun and moon's apparent position against the background stars, and so forth. For convenience, these events are then sub-divided into arbitrary units that make time more measurable: hours in a day, for example, or weeks in a month. In many timekeeping systems, the original observational ties have been weakened or removed over time (for example, in the Gregorian calendar the months are no longer tied to the lunar cycle). In other systems the observational aspects of time remain more pronounced. For example, the Koran specifies exactly how to tell day from night (defining a "day") and solar-lunar calendars (such as the traditional Chinese calendar)

define months in terms of the lunar cycle. Timekeeping units based on observation are sometimes called "observed time" or "wall time" (since a clock and/or calendar mounted on the wall would show that particular time in that particular place).

2.2 Incremental Time

Observed time has many disadvantages computationally. Observed events are not always predictable or convenient to use. The advent of mechanical timekeeping has allowed a different kind of time to flourish: "incremental time" based on a monotonic progression of fixed units. In some cases, incremental time is merely a prediction of when an event might be observed.

Universal Coordinated Time (UTC) is the basis for modern timekeeping: among other things, it provides a common baseline for converting between incremental and observed time. UTC is also known a GMT (Greenwich Mean Time). There are some subtle differences between the two, but none that affect the average person.

2.3 What Is a Time Zone?

A time zone is a set of rules for determining the local observed time (wall time) as it relates to incremental time (as used in most computing systems) for a particular geographical region.

Before the adoption of time zones, local time was derived directly from observation. Clocks might be set, for example, based on an observed event such as local noon. Traveling fairly short distances across the Earth's surface results in changes in local observed time: you only have to travel about 28 kilometers (17 miles) at the equator (and less distance the further north or south you travel) to alter the observed local noon by one minute.

Time zones were originated in several countries by railroad operators. Maintaining a schedule for large geographic areas allowed people in the various locations served by the railroad know when the train would arrive (and depart). Coordinating trains could be scheduled between stations (using a single line in alternating directions, for example), avoiding observational error, local customs, and other issues combined with a plethora of "local times" to make accurate train scheduling of this sort difficult.

Railroads solved this problem by adopting fixed regions in which the same local time was used throughout. These "time zones" were "one hour wide": the local time in the middle of the time zone was used throughout the region, so that the most observational deviation most people would see was about half an hour (and most people experienced a smaller deviation). This is a value small enough that most people won't notice the difference between actual and observed time.

2.4 What about Daylight Saving (Summer) Time?

More recently, the concept of "Daylight Saving Time" (DST) or "Summer Time" was adopted as a way of allowing people more sunlight hours in the evening. DST varies from country to country (not to mention locality-to-locality) and often has "special one-off" changes to accommodate special events. Not all regions observe DST: usually those closer to the equator do not need it.

2.5 How is a Time Zone Defined?

There are different definitions, resulting in different numbers of time zones, depending on which of the following criteria are taken into consideration:

- **UTC Offset** All time zones have, as their basis, an offset from UTC. It is the difference (positive or negative) between when a given time event is observed in UTC and in local time. If all time zones used one-hour offsets, there would be 25 world-wide time zones, ranging between 12 hours before UTC to 12 hours following UTC. However, there are some that use half-hour or even quarter-hour offsets (or even some odd offsets). In addition, some time zones fall outside a single-day span.
- **Observation of DST** Some times zones include rules for observing DST, while others do not. The observation of DST is defined by a set of rules that include:
  - *Amount of DST offset* Amount of time the clock changes when DST starts or ends. Nowadays this is one hour, but other values are theoretically possible (and have been used historically).
  - *Starting and ending day and time of DST* The day and time of day when DST begins and the day and time of day when it ends, which varies by locality. For example, most areas that use DST do so in the summer time. That is, they change their UTC offset forward by one hour when DST starts in the spring and the reverse when DST ends in the autumn. Since "spring" and "autumn" happen in opposite parts of the year in the northern and southern hemispheres, the starting and ending days are very different for zones in opposite hemispheres. But note that even regions that share a UTC offset and are similar in latitude (or are even share borders) may have differing DST start or stop rules.

- **Adoption Dates** Regions that currently have the same UTC offset and DST behavior may have had different rules in the past. Correct handling of past time values requires treating such regions as separate time zones. For example, Korea Standard Time and Japan Standard Time currently use the same UTC offset and neither uses daylight saving. However, Japan abandoned DST in 1951, while South Korea used it last in 1988, so an application that tracks time values that reach back that far might need to track these time zones separately.

  Example 1:

  Adoption dates can be applied to any of the values that define a time zone, such as the amount of DST offset and the starting/ending dates or times for DST. For example, prior to 2007, the United States started DST at 2 a.m. on the first Sunday in April for each time zone observing DST. In 2007, the USA changed the start date and time to 2 a.m. on the second Sunday in March.

3 Representing Time Values in Data Structures

There are several ways to represent time data, which vary in suitability according to application need:

3.1 Incremental Time

Incremental time measures time using fixed integer units that increase monotonically from a specific point in time (called the "epoch"). Most programming languages and operating environments provide or use incremental time for working with time values. Incremental time

is not usually seen directly by users, but is mapped to observed time for human consumption.

Example 2:

The Java type `java.util.Date` is a long (integer) value. It represents the number of milliseconds since midnight (00:00 a.m.) on January 1, 1970 in UTC (less all of the intervening leap seconds). Other systems use different units and/or epochs.

Date and time values based on incremental time are time-zone-independent, since at any given moment it is the same time in UTC everywhere: the values can be transformed for display for any particular time zone offset, but the value itself is not tied to a specific location.

Not all incremental time representations are tied to an epoch. A monotonic clock system might just be tied to the last time the clock was restarted or some other event.

Note: One quirk of timekeeping is the need for leap seconds. The Earth's rotation is not even and is slowing down. To help address this problem, the [International Earth Rotation Service] occasionally mandates a "leap second" to keep the calendar and other field-based time values in sync with incremental timekeeping systems such as atomic clocks. A leap second occurs once or sometimes twice per year and always takes the form of an additional second added to the last minute of the day. Usually the leap second is added to December 31st or June 30th.

As mentioned above, many incremental time values (such as Java's or POSIX's) do not keep track of leap seconds in their incremental time values. What happens is:

1. Eventually, the actual clock is updated externally by the user or via a service such as NTP. Most computer clocks exhibit some amount of clock drift anyway, so this sort of maintenance is not unusual.
2. No list is kept of past or future leap seconds (and no list exists for dates preceding the advent of leap seconds in 1972), so software often doesn't include leap seconds when calculating the difference between two datetimes. For example, the difference between 12:00:00 Noon on December 31st and 12:00:00 Noon on the following January 1st will always be 86400 seconds, even if a leap second was mandated for the intervening midnight.
3. There may be no way to represent a leap second time value using your local incremental units and may not be a means of representing a leap second using field-based units. For example, while Java's `java.util.Calendar` class allows for a "61st" second of a minute to accommodate leap seconds, if you set a Java Calendar to December 31, 2008 23:59:60 UTC (a recent leap second value) and then convert that to a `java.util.Date` in order to print it out, you might see: `"January 1, 2009 00:00:00 UTC"` because the Date doesn't represent leap second values.

If your application cares about or is sensitive to leap seconds, special care must be taken to deal with the loss of leap second precision.

3.2 Field-Based Time

Field-based time divides observed or wall time into separate field values such as year, month, day, hour, minute, second, etc. Field-based times may or may not be tied to either UTC or the local time zone—or may be indeterminate. Field-based times are also typically

tied to a specific calendar (such as the Gregorian calendar). The formats described by the ISO 8601 standard are field-based.

Example 3:

The `tm` structure in the C programming language is one such field-based time. For example:

```
#include <time.h>

#include <stdio.h>

int main(void)

{
  struct tm t;

  time_t t_of_day;

  t.tm_year = 2011-1900; /* 1900 based */

  t.tm_mon = 0;   /* January: zero based! */

  t.tm_mday = 3;  /* the third */

  t.tm_hour = 0;  /* midnight */

  t.tm_min = 0;   /* midnight */

  t.tm_sec = 0;   /* midnight */

  t.tm_isdst = 0; /* no DST currently */

  t_of_day = mktime(&t);  /* convert the tm struct to an incremental time_t */

  printf(ctime(&t_of_day));

  return 0;
}
```

A field-based time may or may not include information about the time zone being used. In a purely numeric representation, such as `time_t`, sometimes only the current UTC offset is provided.

3.3 Floating Time

Some observed time values are not related to a specific moment in incremental time. Instead, they need to be combined with local information to determine a range of acceptable incremental time values. We refer to these sorts of time values as "floating times" because they are not fixed to a specific incremental time value. Floating times are not attached and should never be attached to a particular time zone.

Some examples of floating time events include a user's birth date, a document's publication date, a list of official holidays, or the expiration date for an offer (if not tied explicitly to a time zone).

Example 4:

Suppose that your application delivers newspapers to users. Your application wants to show the publication date of each issue so that, for example, *The Sunday News* is always shown

as being published on a Sunday. The publication date is thus a "floating time" value.

Consider this bit of Java code which is attempting to handle one such date:

```
        final String pubDate = "2011-01-02";
            SimpleDateFormat f = new SimpleDateFormat("yyyy-MM-dd");
            f.setTimeZone(TimeZone.getTimeZone("UTC"));
            Date d = f.parse(pubDate, new ParsePosition(0));

        f = new SimpleDateFormat("E yyyy-MM-dd HH:mm");
            f.setTimeZone(TimeZone.getTimeZone("UTC"));
            System.out.println(f.getTimeZone().getID() + "\t" + f.format(d));

            f.setTimeZone(TimeZone.getTimeZone("America/Los_Angeles"));
            System.out.println(f.getTimeZone().getID() + "\t" + f.format(d));

            f.setTimeZone(TimeZone.getTimeZone("Pacific/Honolulu"));
            System.out.println(f.getTimeZone().getID() + "\t" + f.format(d));

            f.setTimeZone(TimeZone.getTimeZone("Pacific/Kiritimati"));
            System.out.println(f.getTimeZone().getID() + "\t" + f.format(d));
    }
```

When we run the code, here is what the results are:

```
UTC                  Sun 2011-01-02 00:00
America/Los_Angeles  Sat 2011-01-01 16:00
Pacific/Honolulu     Sat 2011-01-01 14:00
Pacific/Kiritimati   Sun 2011-01-02 14:00
```

This code parses a date string into a Date object, which is a type of incremental time. This means that, instead of "floating" as needed between time zones, it has a fixed relationship to UTC and might be used to display the wrong value, depending on the local time offset. In fact, any floating date can stretch over a period of up to fifty hours—beginning with the Pacific/Kiritimati time zone (UTC+14:00) and extending up to the Pacific/Honolulu time zone (UTC-11:00).

Note: There is a UTC-12:00 time zone but it is unihabited.

4 Use Cases: How Time Zones Affect Applications

There are a number of different ways that time zones can affect how time values are stored or processed in an application:

4.1 Timestamps

If your application can accurately generate incremental and/or field-based times based on UTC and the events are not tied to specific local time, all that is needed is the timestamp value itself. That is, if your application never needs to recover what the actual wall time was when event occurred and only cares about relative ordering of events. For example, if you merge log files from many machines together or if you are recording events in a log, a timestamp is perfectly adequate. For these types of time events, an incremental time or a field based time with an offset is all that is needed.

In fact, it is often desirable to normalize time values to UTC (or a specific UTC offset) so that separate series of data can be easily compared and merged. Information about local offset may be valuable in recovering the actual wall time, but time zone rules are probably only rarely interesting.

4.2 Past Events

For events that occurred in the past (with no future events) for which you need to know what the wall time was, the time zone of the event may be necessary additional data. Once an event is in the past, its relationship to incremental time becomes fixed and the rules for generating wall time remain static essentially forever. You might still need to know that an event occurred at 10:00 rather than at 14:00 local time. At a minimum, the zone offset from UTC is necessary, although knowing the complete time zone is necessary for some applications. Knowing the specific time zone allows one to reconstruct the time and its relationship to other wall times.

4.3 Past and Future Events

If your application deals with both past and future events (for example, if you have a calendar or a meeting schedule), you'll need additional time zone information to ensure proper time computation. At a minimum you will need the time zone, not merely an offset from UTC. This is because a future event's wall time depends on time zone related information, such as DST transitions. One issue with future events is that time zone rules can change from time to time and these may require an application to update affected data records in order to meet user's expectations. This is because many systems actually store the time portion of the value as an incremental time and the incremental time needs to be changed if the wall time offset from UTC has been altered.

4.4 Recurring Events

A recurring event, such as a regular meeting, is usually defined by a set of rules that express a user's intent. In some cases, the user intends for the event to recur at a specific local time (and thus, wants to tie it to local time changes, such as DST transitions). In other

cases, the user wants the time tied to another time zone, to a specific UTC offset, or to other events. So, for example, a recurring weekly event might need to add 167, 168, or 169 hours to "last week's occurrence" of an event to compute this week's start time, depending on whether a DST transition has occurred and which direction the transition was in.

4.5 Floating Time Events

Floating time events are sometimes indicated by omitting the time zone or UTC offset from a time value. It's generally important to know when a time value represents a floating time, because your application's handling of the value needs to be different.

For example, if your application converts a floating time value of `2009-01-01` to an incremental time with a UTC offset of zero, users examining the value later might see `2008-12-31` because their local UTC offset causes the value to be converted improperly. This is because the incremental time value is expected to fall between `2009-01-01 00:00:00` and `2009-01-01 24:00:00`, local time. The UTC value may or may not fall into this range, depending on the local time offset from UTC.

Since floating time values are often dates without any associated hours, minutes, or seconds, the resulting incremental time for these fields is often set to zero, exacerbating the problem: all time zones west of the prime meridian will consider a floating time to be the previous day.

Consider the Java in Example 5:

Example 5:

```
     Calendar c = new GregorianCalendar();
     c.setTimeZone(TimeZone.getTimeZone("UTC"));
c.clear();                  // clears all fields in the calendar
c.set(Calendar.YEAR, 2009);
c.set(Calendar.MONTH, 8);    // recall that Java months are zero based
c.set(Calendar.DATE, 1);
c.set(Calendar.HOUR, 0);
c.set(Calendar.MINUTE, 0);
c.set(Calendar.SECOND, 0);
long start = c.getTime().getTime();
c.roll(Calendar.DATE, true); // start of the next day
long end = c.getTime().getTime();
TimeZone.setDefault(TimeZone.getTimeZone("America/Los_Angeles"));
System.out.println(Long.toHexString(start) + " " +
 new Date(start) + "\n" +
    Long.toHexString(end) + " " +
    new Date(end));
```

You might expect to see 2009-09-01 consistently, but here's the actual output:

```
12372e6a000 Mon Aug 31 17:00:00 PDT 2009
123780cfc00 Tue Sep 01 17:00:00 PDT 2009
```

5 Representing Time Zones and Zone Offsets

Depending on the types of data your application needs to represent, you may need to alter or extend your data structures to handle the different requirements laid out above.

Example 6:

[XML Schema] follows the ISO 8601 standard for its lexical representation. ISO 8601 is a field-based way to represent time values and increments. A time value can include (or omit) the zone offset from UTC. Howeve, a zone offset is not the same thing as a time zone and the difference can be important depending on your application.

If your application needed to represent a future recurring event, such as the time of a regular teleconference, an `xs:dateTime` value such as `"2010-07-10T07:00:00-07:00"` with an associated xs:duration field of "`P7D`" (i.e. weekly) would not tell you if the meeting should occur at a different actual wall time following the next daylight saving transition, and, if so, which time zone's rules should be applied in order to determine the changes. If the value "-07:00" is supposed to represent the U.S. Pacific time zone, then reminder messages might be generated for the wrong time once U.S. Pacific time transitions to standard time in the fall. In addition, users in other time zones, where the DST transition occurs on a different date, may find themselves unsure of when to call in to such a meeting.

Note that, although ISO 8601 is expressed in terms of the Gregorian calendar, it can be used to represent values in any calendar system. The presentation of date and time values to end users using different calendar and timekeeping systems is separate from the lexical representation, as the time value can be converted to an incremental time and then new field values computed for some other calendar or time keeping system that uses alternate rules.

5.1 Time Zone Identifiers

The most definitive reference for identifying sets of time zone rules is the TZ database (also known as the "Olson time zone database" [tzinfo]), which is used by systems such as various commercial UNIX operating systems, Linux, Java, CLDR, ICU, and many other systems and libraries. Other systems exist: for example, Microsoft Windows uses its own data set and identifiers.

In the TZ database, time zones are given IDs that typically consist of a region and exemplar city. An exemplar city is a city in the time zone in question that should be well-known to people using the time zone. For example, the U.S. Pacific time zone has a TZ database ID of "America/Los_Angeles". The TZ database also supplies aliases for many IDs; for example, "`Asia/Ulan Bator`" is equivalent to "`Asia/Ulaanbaatar`". The Common Locale Data Repository [CLDR] can be used to provide a localized form for the IDs: see Appendix J in [UAX 35]. Note: some systems, such as Apple Inc.'s MacOS, provide additional exemplar cities.

5.2 Identifying Zone from Offset and Country

Most countries are either small enough in area or, for practical reasons, choose to observe only a single time zone for the entire country. This means that knowing the country of the user is frequently sufficient to identify the time zone of the user as well. At the time this document was published, only twenty countries had more than one observed time zone. These countries are: Argentina, Australia, Brazil, Canada, Chile, Democratic Republic of the Congo, Ecuador, France, Greenland, Indonesia, Kazakhstan, Kiribati, Mexico, Micronesia, Mongolia, New Zealand, Portugal, Russia, Spain, and the United States.

Some special cases exist within this list:

- *Countries with maritime or overseas possessions* Chile, Ecuador, France, New Zealand, and Portugal each have islands or other wide-ranging geographic areas far

from the main part of the country. For example, Easter Island is part of Chile, the Galapagos Islands are part of Ecuador, and the Azores are part of Portugal. These offshore possessions are the source of additional time zones in each of these countries.

- *France* France is a special case of the above. There are several regions that are part of France from a legal perspective (although each of them has its own ISO 3166-1 code). These include Reunion Island and French Guiana. Additionally, French Polynesia is divided into three time zones.

Within each of the countries that observe multiple time zones, knowing the current offset and current time will usually allow you to determine the time zone accurately. An exception to this is the United States: there exist some regions, such as Arizona, whose time zone cannot be determined strictly from the current time, country and offset, although an inferred time zone will always work for current time applications (not future and past times).

 5.3 Incremental versus Field-Based Time

Incremental time and field-based time differ in the way certain operations work. For example, incremental times can be directly compared: their integer values determine which is earlier or later. Field based times must be normalized and their individual fields compared. Field based times can have certain kinds of logical operations performed on them (for example, rolling the month forward or back), while incremental time requires a logical transformation.

For example, to set the date `2005-08-30` forward by one day, an implementation can add 'one unit' to the "day" field and adjust the month and year as appropriate. In incremental time, a similar operation might be performed by incrementing the value by 24 hours * 60 minutes * 60 seconds * 1000 milliseconds, which is one logical day. However, there may be errors when a particular day has more or fewer seconds in it (such as occur during daylight saving transitions).

Bear in mind that rolling fields forwards or backwards in field-based times can be tricky. For example, Feburary does not always have the same number of days in it. Or consider the problem of incrementing the month forward by one in the date `2011-01-30`.

The SQL data types `date`, `time`, and `timestamp` are field based time values which are intended to be zone offset independent: they are actually, technically, floating time values! The data type `timestamp with time zone` is the zone offset-dependent equivalent of `timestamp` in SQL. Programming languages, by contrast, tend to use incremental time and convert to and from a localized textual representation on demand. Databases may use incremental time or either zone offset-dependent or independent field-based structures internally. For example, Oracle databases treat a `timestamp` field as though it is in the local time of the database instance. This can have unusual effects on queries: a field based time value that represents a local time with daylight saving needs to identify the UTC offset and whether daylight saving is in effect or not in order to disambiguate the repeated time when transitioning from daylight saving time (DST) to standard time. For example, in the United States in 2009, any instant between 1 and 2 am on November 1st happens twice, once in DST and a second time in standard time. Field based types without a time zone field (such as an Oracle timestamp) cannot represent the repeated times unambiguously without supplemental information provided externally.

As a result, users may not be clear on the differences between these types or may create a mixture of different representations. For example, a Java programmer using JDBC will

retrieve incremental times (`java.util.Date` objects) from a database, even though the actual field in the database is a (field-based) timestamp value.

In XML Schema, as with SQL, dates and times are always expressed using field-based time. The date or time may express the zone offset from UTC (for example using a format such as `08:00:00+01:00`). UTC is indicated by the letter Z (for example `08:00:00Z`). Or, the zone offset may be omitted completely.

Properly speaking, an XML Schema date or time value with a zone offset is field-based/zone offset dependent and one without is field-based/zone offset independent.

If the two types are mixed, then the interpretation of the zone offset is not adequately specified in XML Schema. In XQuery 1.0 and XPath 2.0 Functions and Operators [XPathFO], the interpretation is implementation-defined and is based on an implicit zone offset. This is usually either UTC or local time. The presence or absence of the zone offset in the XML Schema representation may not be indicative of the original data's intention because of the confusion described above. Proper comparisons or processing rely on normalizing all date and time values into zone offset-independent (or zone offset-dependent) forms and never mixing the two in a particular operation.

6 Guidelines

This section describes different guidelines that can be applied to various time and date comparisons.

6.1 Guidelines Summary

Generally:

- When creating content, use UTC for your time values whenever possible so that values from discrete sources can be compared more readily

If you are using incremental time:

- Decide if the value is time zone dependent or time zone independent.

If you are using time zone independant values:

- Omit zone offset
- Ignore zone offset on values received
- If a zone offset is required, use UTC (offset = 0)

If you are using time zone dependent values:

- Pass both offset and zone ID if appropriate
- Specify the actual time zone, preferably using the Olson ID, in an additional data field and use this value to compute the incremental time
- Use the time zone (or time zone offset if the former is not available) to compute the canonical date value.
    - If no time zone or time zone offset information is available, use UTC (0) as the offset when converting to an incremental time value
    - Always specify the zone offset in formats that permit it (such as HTML5 or XML Schema)

    - Use UTC as the offset whenever possible (HTML5: global date and time with time zone of 'Z')
    - Use the actual zone offset if UTC cannot be used.
    - Be sure to use the correct offset for that date and time and not just the current offset in that time zone or the raw offset of that time zone. For example, if a system in the U.S. Pacific time zone (America/Los_Angeles) generates a dateTime value `2005-02-11T11:23:04-07:00` on `2005-08-16`, it may be an error (since the offset from UTC during August in that time zone is UTC-7, but the zone offset in February is UTC-8).
- Treat values with no time zone or time zone offset as if they all use the same offset.
- UTC (offset 0) or local time are two potential good choices in this case
- For time values with no date: indicate the zone offset if the offset value is fixed. Supply additional data fields if it is not. For example, this would not apply to a meeting scheduled in U.S. Pacific Time, but would apply to a meeting that is always UTC-08:00 (and thus at 7:00 in the morning in U.S. Pacific Time during parts of the year).
- Use incremental time values to avoid comparing across time zones if possible
- Compare time values using canonicalized time zones
- When comparing times, floating times with no time zone information should use UTC as the time zone.
- When comparing local times, use zone offset information to canonicalize the values.
    - Avoid using local times if possible.

If you are working with repeating events:

- Store originating offset and whether to apply DST rules in addition to time and time zone
- Recompute future incremental times if time zone rules change (you'll need the original offset to do this)

If you are selecting or negotiating time zone:

- Allow the user to choose a time zone associated with a user session or profile if possible.
    - Consider using exemplar cities to help users identify the time zone.
    - Use the country as a hint, since most countries have only a single time zone
    - Omit historical time zones if appropriate
- use IP-geolocation, cellular radio country code, GPS data, or other external data sources if available.
- Use an explicit zone offset with date, time, and dateTime types, if possible.
    - Include an additional field indicating the time zone, if possible.
- Avoid applying operations based on date or time types (such as indexing) to collections of data in which some data items may have zone offset information and other data items may not have zone offset information.
- If you have data that includes implicit and fixed explicit zone offsets, before applying any date- or time-sensitive operations adjust the zone offset of the implicit data to UTC with the functions for zone offset adjustment, cf. sec. 10.7 in [XPathFO].
- If you have data that contains both implicit and fixed explicit time zones and you do not want to adjust the data subset which already has a zone offset, make sure that you recognize this data subset, for example via the component extraction functions, cf. sec. 10.5 in [XPathFO].

6.2 Working with Future and Recurring Events

When creating an application that can store values in the future, including recurring events, you'll need to make additional data fields to ensure that you can reconstruct the user's intentions and adapt future time values to changes in time zones and time zone rules (especially alteration of daylight saving/summer time start and stop).

If the time should change based on the rules for a given time zone (such as DST transition), store the originating time zone and original offset applied. If the event is recurring, you must also store a flag indicating whether DST transitions should be applied to future occurrences of the event or not: this tells you whether the user intended a specific wall time or a specific incremental time. For example, if you schedule a phone call for Friday, August 27, 2010 at 10:30 Pacific Daylight Time, then the field values stored might be:

```
2010-08-27T10:30:00-07:00  !-- -07:00 is the originating offset value
America/Los_Angeles        !-- This is Pacific time
false                      !-- don't alter the time when DST changes or rules change
```

If you then set up a recurrence rule, such as "weekly" for this call, you can compute that "10:30 Pacific" becomes "9:30 Pacific" when the Pacific time zone transitions back to standard time from summer time. If the rules change for the Pacific time zone, you won't need to alter your data or search through all records in you database to update the data (you will still have to recompute the incremental time, though). When setting up recurrence:

- Store the originating offset and whether to apply DST rules in addition to time and time zone
- Recompute future incremental times if time zone rules change (you'll need the original offset to do this)

 6.3 Negotiating Time Zone with the User

Users may sometimes need to specify time values, such as in an HTML form. If the time is the current time, then ordinary incremental time may be used ("it's the same time everywhere in UTC"). However, if the user is specifying a time or date in the future or past, the time zone being used becomes important.

It may be possible to determine the user's current time zone from the browser. For example, converting an array of date values to local time can be used to determine the UTC offset and daylight saving transition (if any), which can then be compared to known rules for time zones. In other cases, external data such as IP-geolocation, cellular radio country code (ITU E.212 MCC), or GPS data may be available. But for most systems, the user will need to choose a time zone. There are always edge cases in which even very good external data cannot resolve the time zone accurately.

Because there are many time zones in the world, one way to make choosing time zone more accessible is to have the user to choose country first. For the few ambiguous cases, the user can then be presented with a much smaller number of specific choices. Note that the Olson time zone database also defines a large number of zones that are of mainly historical interest (their rules were different from current time zone rules in the past but the zones are no longer distinct). These historical zones should usually be omitted from any user choice for time zone.

- Use incremental time in UTC if possible (HTML5: global date and time with time zone of 'Z')
- Allow the user to choose a time zone associated with a user session or profile if possible.

    - Consider using exemplar cities to help users identify the time zone.
    - Use the country as a hint, since most countries have only a single time zone
    - Omit historical time zones if appropriate
- use IP-geolocation, cellular radio country code, GPS data, or other external data sources if available.

 6.4 Working with Field-Based Dates and Times based on XML Schema

Field-based time and date values require the user to determine whether to use a fixed zone offset, a time zone, or nothing. While XML Schema times are field-based in terms of the lexical representation, the underlying data or implementation may use incremental time, as may the implementation processing the values. Each specific case requires specific handling.

For incremental time values: use a specific zone offset, preferably always UTC

- UTC is strongly recommended as this offset, since most incremental time systems are based on it
- Values that do not specify a zone offset should be treated as if they use the same offset. If UTC is used, this produces the least amount of modification in the data.
- For time zone independent values (such as a list of employee birthdays): Omit the zone offset.
    - When processing, ignore any zone offset values in these fields.
    - When storing, strip off any zone offset values since zone changes are probably an artifact of other processing.
    - If a zone offset is required, use UTC
- For time zone dependent values: always supply the zone offset.
    - Be sure to use the correct offset for that date and time and not just the current offset in that time zone or the raw offset of that time zone. For example, if a system in the U.S. Pacific time zone (`America/Los_Angeles`) generates a `dateTime` value `2005-02-11T11:23:04-07:00` on 2005-08-16, it may be an error (since the offset from UTC during August in that time zone is UTC-7, but the zone offset in February is UTC-8).
- For time values with no date: indicate the zone offset if the offset value is fixed. Supply additional data fields if it is not.
    - That is, this would not apply to a meeting scheduled in U.S. Pacific Time, but would apply to a meeting that is always UTC-08:00 (and thus at 7:00 in the morning in U.S. Pacific Time during parts of the year).

 6.5 Working with Date and Time Values that Require a Time Zone (and not a zone offset)

Documents or systems can also choose to accompany a time value with the appropriate time zone identifier or TZID using a complex type. This is very important with recurring times, such as calendar meeting times. If a regular meeting is at "08:00 Pacific Time", it is insufficient to store and interchange just a zone offset.

Unfortunately, XML Schema date and time types do not provide for Olson IDs, so most time operations cannot use TZIDs directly. Time zone identification in the date and time types relies entirely on time zone offset from UTC. It is up to the document designer to keep the TZID in a separate data field from the time value.

There are different ways to compare two `<datetime, TZID>` pairs. If both the date and time are fixed (`2004-09-31T01:30`), then this can be done by computing the offsets on that date and at those times, using the TZ database. This order then reflects whether one datetime is (absolutely) before another.

If the dates are not fixed (such as `<T01:30, TZID>` — notice that the date value is omitted) then in some sense, neither is 'before' the other, since each refers to a repeating, interleaved set of points in time. The simplest comparison mechanism where the dates may not be fully specified is simply to put both in canonical form, then order them first by time then by TZID (alphabetical, caseless order). The Olson database does not maintain a fixed canonical form; however, CLDR does provide such a form. (It is also possible to have a looser comparison, whereby `<time0, TZID0>` is compared to `<time1, TZID1>` over some interval of time: if one consistently has a smaller offset during that period, it is considered to be less than the other value. However, there are cases where this mechanism results in a partial ordering.)

Note that when you pass a dateTime and time zone ID together, you may wish to supply the time zone's offset as part of the data (using local time rather than UTC) so that applications have access to both sets of information. For example `<T01:30-08:00, "America/Los_Angeles">` tells you it is both the US Pacific time zone and that the offset was 8 hours from UTC (making it standard time, but also making it 9:30 UTC).

A different example would be an airline reservation system. These normally use local time at originating and destination airports to express time values. That is, you can have a flight that leaves one airport at 12:00-04:00 and arrives the same day at 12:00-07:00 and takes three hours in the air.

- Use incremental time values to avoid comparing across time zones if possible
- Use time zone offsets to compute the canonical date value.
  - Pass time zone as an additional data field.
  - Use Olson zone IDs if possible
  - Path both offset and zone ID if appropriate
- Compare time values using canonicalized time zones

7 Comparing Times

Conversion between or operations on data sets that mix values with and without zone offsets present certain problems.

Example 7: Values with and without zone offsets

```
<aDateTime>2005-06-07T13:14:27Z</aDateTime>
<bDateTime>2005-06-07T11:00:00</bDateTime>
```

If one wishes to write a comparison between the value of `<aDateTime>` and `<bDateTime>`, then the two values must be reconciled to use the same reference point. `<aDateTime>` uses UTC and can easily be converted to computer time or shifted to another zone offset. `<bDateTime>` contains no indication of the zone offset. It may be UTC or any other value (currently up to 14 hours different in either direction from UTC).

It is good practice to use an explicit zone offset wherever possible. If one is not available, best practice is to use UTC as the implicit zone offset for conversions of this nature. This is because the values are exactly centered in the range of possibilities and because representation internally (as computer time) is usually based on UTC. Since a single

reference point has been used it may be possible to unwind the change later even if erroneous conversion takes place. When working with multiple documents from various sources, the "implicit" offset of the document may vary widely from that of the implementation doing the processing. If UTC is widely used, the chances of error are reduced.

Content and query authors are warned that comparing or processing dateTime values with and without time offsets may produce odd results and such processing should be avoided whenever possible. Generating content that omits zone offset information (where it exists) is a recipe for errors later. Of course, data such as the SQL types cited earlier and which are meant to represent wall time, should continue to omit the zone offset. Query writers can check for the presence (or absence) of zone offset and should do so to modify dates and times explicitly (instead of allowing implicit conversion) whenever possible.

- When comparing times, floating times with no time zone information should use UTC as the time zone.
- When comparing local times, use zone offset information to canonicalize the values.
  - Avoid using local times if possible.
  - If no offset information is available for a given document or field, use UTC for that value
  - When creating content, use UTC for your time values whenever possible so that values from discrete sources can be compared more readily

 7.1 Working with XQuery / XSLT

Users of XQuery 1.0 and XSLT 2.0 and other standards should take the following recommendations into account even if time-zone-sensitive data is rarely used. Sooner or later some data will be affected by the issues described:

- Use an explicit zone offset with `date`, `time`, and `dateTime` types, if possible.
  - Include an additional field indicating the time zone, if possible.
- Do not apply operations based on date or time types (such as indexing) to collections of data in which some data items may have zone offset information and other data items may not have zone offset information.
- If you have data that includes implicit and fixed explicit zone offsets, before applying any date- or time-sensitive operations adjust the zone offset of the implicit data to UTC with the functions for zone offset adjustment, cf. sec. 10.7 in [XPathFO].
- If you have data that contains both implicit and fixed explicit time zones and you do not want to adjust the data subset which already has a zone offset, make sure that you recognize this data subset, for example via the component extraction functions, cf. sec. 10.5 in [XPathFO].

Acknowledgments

This document is based on several previous documents. The original Working Group Note (Working With Timezones) was written by Martin Dürst, Mark Davis, Felix Sasaki, and Addison Phillips. Portions of this document, notably the introduction, were taken from an older document ("It's about time") by Addison Phillips. Information on time zone scenarios is based on work by Norbert Lindenberg.