RECURSION SOFTWARE, Inc.

**Software** for Software Developers™

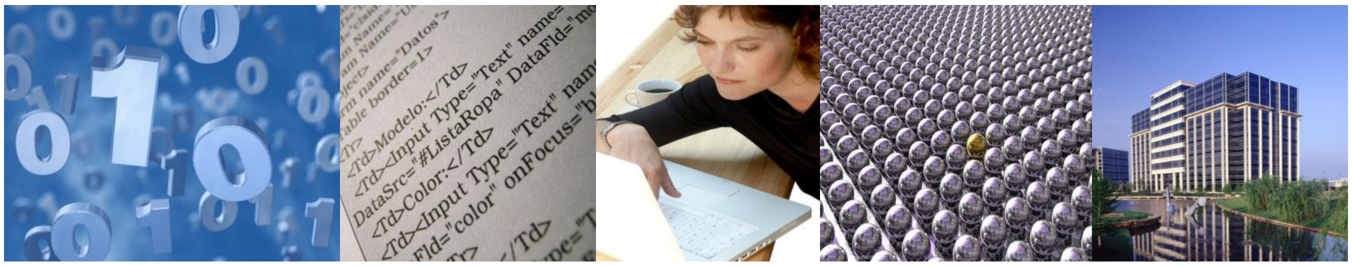
C++ toolkits™

# Time Toolkit: Portable, Flexible Software

By Katherine Ye, Senior Software Engineer

September 6, 2011

## Recursion Software, Inc.



## TABLE OF CONTENTS

# Time Library: A Great Tool to Manage Time and Date

By Katherine Ye

**Abstract**

Time and date calculation is essential to application development. Even though it sounds simple, it can often be quite complex. This article explores a set of APIs (Application Programming Interface) offered by Recursion's Time Toolkit that greatly reduces the complexity of handling issues such as time zones and daylight saving time. This can be especially valuable for businesses that have multiple customer bases, operational bases, or suppliers around the world and that need an accurate and efficient way to handle date and time.

## Why we need time library

In many situations in business and life, a simple record of time and date is not sufficient. You want to know how many days, hours, minutes, seconds, or even microseconds have elapsed since a special event. You need time and date calculation. You can argue that it is just simple math. But is it?

## Time and time zone

### 1. Impact of time zone and daylight saving time

Let us use the United States as an example. The United States uses nine standard time zones: Atlantic Standard Time (AST), Eastern Standard Time (EST), Central Standard Time (CST), Mountain Standard Time (MST), Pacific Standard Time (PST), Alaskan Standard Time (AKST), Hawaii-Aleutian Standard Time (HST), Samoa Standard Time (UTC-11) and Chamorro Standard Time (UTC+10).  Not all states in the U.S. support daylight saving time. Puerto Rico, Hawaii, U.S Virgin Islands and American Samoa do not, and neither does Arizona except in the Navajo Indian Nation. Following is the time zone map for the U. S.

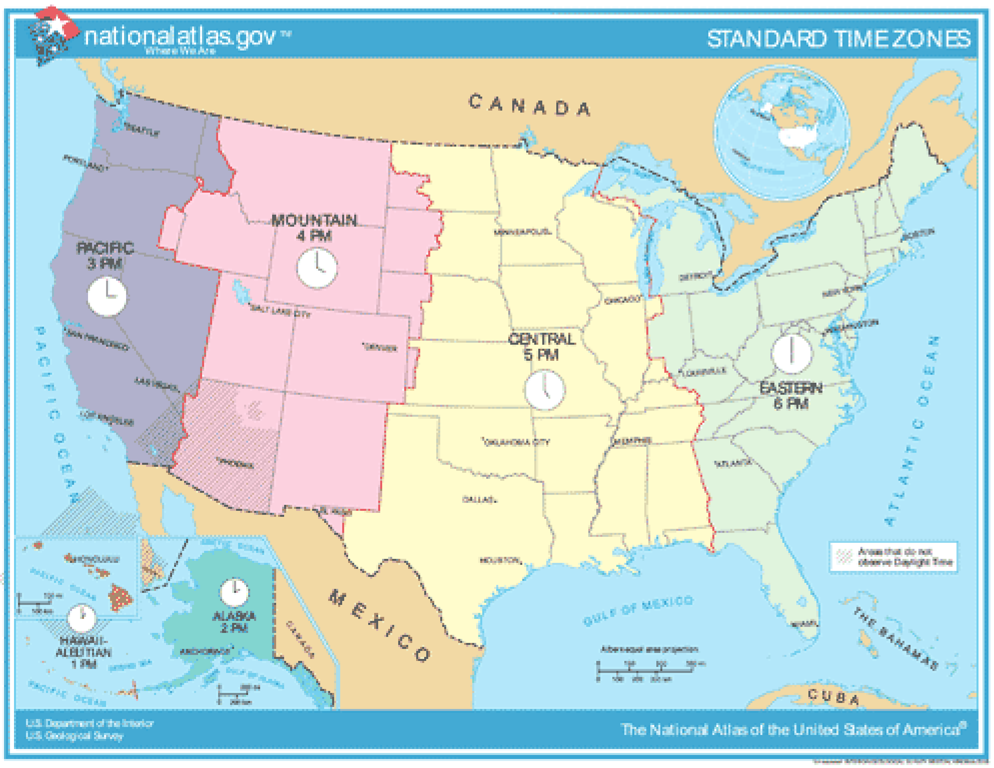


Fig 1. US time zone map.

Even though calculation of time and date seems plain and simple, daylight saving time can inadvertently complicate a seemly simple calculation. For example, let us assume a start date of Oct 12, 2008. Add 23 days to it. The result, you'll probably say, is Nov 4, 2008. But the real answer is that it depends. It depends on *when* you start counting on Oct 12, 2008. The result can be either Nov 3, 2008 or Nov 4, 2008. Daylight saving is the reason for this ambiguity.

Daylight saving time in the U.S. in 2008 started on Sunday, March 9 and ended on Sunday, November 2. In the example, the start date of 10/12/2008 is in the daylight saving time. But 23 days later, it is no longer in the daylight saving time. The clock had been turned back by one hour. If we start from Oct 12, 2008 00:06:30.000000 and add 23 days, the result is Nov 4, 2008 00:05:30.000000. But if we start calculating from Oct 12, 2008 00:00:00.000000, the result is Nov 3, 2008 23:00:00.000000. In the first instance, because of the daylight saving time, the time moved back from 6:30 to 5:30. It is one hour short. In the second instance, daylight saving not only changed the time but also impacted the date. As you can see, a simple date calculation can end up with an interesting twist.

This is just a sample illustration of daylight saving time's potential impact on date calculation. Things can become more complicated if the calculation involves different parts of the world and different time zones. To make things worse, not every country observes daylight saving time. Among those that do, the start and end dates vary. Some of these countries have adopted daylight saving time only a few years ago, while others have had it for decades. If we consider all the variants, time and date calculation could be a daunting task even within a small region.

What we need is a Time Library that has the ability to compare dates and times, add time durations, retrieve dates and times, and work naturally with date and time intervals. Ideally, the library should have a simple set of APIs that makes time and date calculation just like programming with numbers. The library will not only handle different time zone rules, different daylight saving times, and fundamental time and date infrastructure, but will also manage complex date and time calculation, formatting, and conversion. It will let application developers focus on their business logic and not worry about adjusting underlying details of time and date.

Here we introduce a powerful and portable Time Library: Recursion Software's Time Toolkit.

## 2. Benefits of Recursion's Time Toolkit

Recursion's Time Toolkit has a rich set of time and date abstractions. It supports arithmetic calculation for date, time, time and date, and time period. It provides a robust set of operators. It automatically adjusts time and date according to time zone and daylight saving time. It also has an abstract base class time_zone_rule which can be used to customize time zone.

The benefits of Time Toolkit include, but are not limited to:

- Provide concrete classes for manipulation of dates and times
    - Compare, subtract, add, display dates for a variety of useful information
- Provide concrete classes for manipulation of times
    - Represents microsecond accuracy time between 00:00:00.000000 and 23:59:59.999999
    - Compare, subtract, add, display time for a variety of useful information
- Provide a basis for performing efficient time and date calculations
    - Days between dates
    - Durations of times
    - Durations of dates and times together
    - Ability to handle cross time-zone issues
    - Adjustments for daylight saving time
- Provide comprehensive time zone functionality and predefined rules on time zones and daylight saving time
- Contain extensible time zone rules, providing easy adaptation to changes in regional policies
- Multi-platform support; runs on
    - Linux
    - Aix
    - HP-UX
    - Solaris
    - Mac OS X

- Windows
- Symbian
- Texas Instruments TMS320DDM6437 DSP

Time Toolkit offers the following classes:

- os_date – by using a 64-bit range, time and date can be represented from Jan. 1, 4713 b.c. to Dec 31, 32766 a.d.
- os_time -- a time between 00:00:00 and 23:59:59. Unlike older libraries accurate only to the second, Time Toolkit provides precision to the microsecond.
- os_time_and_date -- a time and date between 00:00:00 Jan. 1, 4713 b.c. and 23:59:59 Dec. 31, 32766 a.d.
- os_time_period -- a period between two points in time.
- os_stopwatch -- measures elapsed real time.
- os_time_zone -- a time zone, including offset from Greenwich Mean Time (gmt) and Daylight Saving Time (dst) rules.
- os_time_zone_rule -- the abstract base class of all time zone rules.
- os_calendar_date -- flexible way to specify a particular day of the year.
- os_simple_time_zone_rule -- adds an offset if a time lies in a certain range, such as Daylight Saving Time (dst) rules.

## Programming with Time Toolkit

Time Toolkit has a complete set of interfaces to handle time and date calculation, time zone rules, and daylight saving adjustments. It is also portable. It behaves the same regardless of the underlying operating system. Programs developed using Time Toolkit work on different platforms and compilers. It has the same code base and the same look and feel across a variety of platforms such as Windows, Linux, AIX, HP-UX, Solaris, and Mac OS X. It simplifies the process of time related programming and can be used and re-used in multiple applications, from the most simple to the highly complex. It allows application developers to focus on the development of the core of the application and to save time by not re-implementing low-level functions.

Another huge benefit of Time Toolkit is that its components are available as source code. This allows application programmers to rebuild libraries or to extend their feature set so as to tailor the Time Toolkit for the special needs of their applications.

Time Toolkit contains classes and methods exposing simple APIs for building portable time and date applications quickly. It is easy to use. All you need to do is call the Time Library initialize routine. Depending on your application needs, you can use overloaded operators to perform calculations.

The following example determines the period between two dates: today and the day a person named Peter was born. It demonstrates several basic features, such as comparing two dates, accessing a date's attributes, and performing arithmetic on a date.

List 1 illustrates accessing date attribute and date calculation

```
#include <iostream>
#include <ospace/time.h>

void
main()
  {
  // toolkit initialization
  os_time_toolkit initialize;

  // construct Peter’s birth date
  os_date born( os_date::april, 8, 1970 );


  // calculate current date
  os_date today = os_date::today();

  // calculate tomorrow
  os_date tomorrow = today + os_time_period( 1, 0, 0, 0, 0 );
  cout << "born = " << born << " today = " << today << endl;
  cout << "tomorrow = " << tomorrow << endl;
  cout << "today > born = " << (today > born) << endl; // Compare
  cout << "born.year() = " << born.year() << endl;
  cout << "born.is_leap_year() = " << born.is_leap_year() << endl;
  cout << "born month = ";
  cout << os_date::month_name( born.month () ) << endl;
  cout << "born weekday = ";
  cout << os_date::weekday_name( born.weekday () ) << endl;
  cout << "days in 1970 = " << os_date::days_in_year( 1970 ) << endl;

  // convert to days.
  os_time_period period = today - born; // Subtract.
  cout << "days alive = " << period.to_days() << endl;
  }
```

Output:

```
born = 04/08/70 today = 05/27/11
tomorrow = 05/28/11
today > born = 1
born.year() = 1970
born.is_leap_year() = 0
born month = April
born weekday = Wednesday
```

days in 1970 = 365
days alive = 15024

List 1. Access date attribute and date calculation

## Local time adjustments

Local time adjustment can be complex and hard to deal with. This is especially true if the time and date calculation involves different time zones as well as different daylight saving times. The rules associated with these adjustments are needed in many applications. Fortunately Recursion's Time Toolkit provides a simple set of APIs which can solve the problem easily.

Let us consider the following scenario.

An airline company has a project that requires calculation of destination date and time based on departure time and flight duration. Since the original location and destination may belong to different time zones and daylight saving time may end after the airplane departs the original region, it is probably not wise to simply add flight duration to the start date and time. To get the accurate time and date at the destination, we need to adjust for time zone and daylight saving time.

As we have mentioned before, Time Toolkit provides a set of APIs which handles the logic associated with time zone and daylight saving transitions. To calculate destination time and date, all the application developer needs to do is initialize the library, construct departure time, add flight duration to departure, set time zone to the destination, and display the correct arrival time.

The following code snippet illustrates the scenario.

```
#include <iostream>
#include <ospace/time.h>

void
main()
 {

  // toolkit initialization
  os_time_toolkit initialize;

  os_time_and_date::default_format( "%B %d %y (%Z), %H:%M" );

  // departure from Los Angeles on Nov 5, 2011 at 10:40 pm
  os_time_and_date departure_time_lax
    (
    // time set to 10:40 pm
    os_time( 22, 40 ),
    // date set to on Nov 5, 2011
    os_date( os_date::november, 5, 2011 ),
    // Los Angeles belongs to pacific time zone.
    os_time_zone::pacific()
    );
  cout << "departure time in LA = " << departure_time_lax << endl;
  cout << "time zone in LA = " << departure_time_lax.time_zone() << endl;

  // flight's duration 4 hours and 50 minutes
  os_time_period duration( 0, 4, 50, 0, 0 );

  os_time_and_date arrival_time = departure_time_lax + duration;

  // Change to eastern time zone. It is the time zone where Miami
  // International Airport belongs
  arrival_time.time_zone( os_time_zone::eastern() );

  cout << "arrival in MIA = " << arrival_time << endl;
  cout << "arrival Time MIA = " << arrival_time.time_zone() << endl;

  return 0;

  }
```

List 2. Arrival time calculation with different time zone and daylight saving time

Output:

```
departure time in LA = November 05 11 (PDT7), 22:40
time zone in LA = os_time_zone( PST8, PDT7 )
arrival in MIA = November 06 11 (EST5), 05:30
arrival Time MIA = os_time_zone( EST5, EDT4 )
```

In the above example, an airplane is scheduled to depart from Los Angeles (Pacific Region) to Miami (Eastern Region). For most Americans, daylight saving time in 2011 starts at 2 a.m. on Sunday, March 13, when most states spring forward by an hour. Time falls back to standard time again on Sunday, November 6, 2011 at 2:00 am; clocks are turned back by one hour and daylight saving time ends.

Fig 2 and Fig 3 illustrate clock movement when daylight saving begins on Mar 13 and ends on Nov 6.

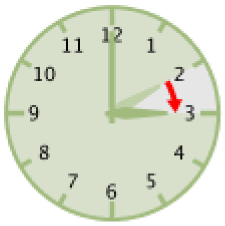


Fig 2. Daylight saving begins

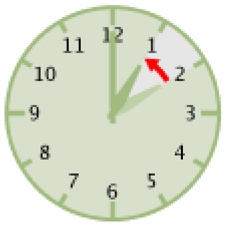


Fig 3. Daylight saving ends

The example in List 2 not only illustrates how Time Toolkit handles transitions over different regions (Pacific to Eastern Region) but also demonstrates how it deals with transition of daylight saving time. The airplane leaves Los Angles on Nov 5 at 22:40 pm, which is still in daylight saving time. Since Miami's daylight saving time ends at 2:00 a.m. in the middle of the flight, the clock moves back by an hour. Thus, after a flight of 4 hours and 50 minutes, the plane arrives in Miami on Nov 6 at 5:30 a.m.

Keeping track of time zones of different regions *and* daylight saving time could be a real headache for an application developer. But with the help of Time Toolkit, the calculation becomes straightforward:

1. Construct the departure time
2. Calculate arrival time by adding flight duration to the departure time
3. Set arrival time zone into destination time zone
4. Display the correct arrival time (with time zone and daylight saving adjustment).

It hides all the complexities of adjusting time zone and daylight saving behind the scene.

## Formats

When it comes to time and date displays, different applications have different needs. Some require microsecond precision, while others may require that the time zone be displayed along with date. With the help of Time Toolkit, you can specify the particular format for the desired output for your time and date application. Time Toolkit has a wide range of formats for time and date representation. The `os_date`, `os_time`, and `os_time_and_date` classes use a format specifier when an instance of the class is streamed. You can specify the format by using `to_string()` or by calling `default_format()`. Each class defines `default_format()` access functions that access and modify its format specifier.

The following example demonstrates how to get the desired time and date output by using format string.

```
#include <iostream>
#include <ospace/time.h>

void
main()
 {

  // toolkit initialization
  os_time_toolkit initialize;

  // construct a time
  os_time time( 11, 4, 22 );
  // use to_string to set format, where
  // %I -- two-digit hour (01 to 12)
  // %M -- minute (00 to 59)
  // %S -- seconds (00 to 59)
  // %p -- locale's equivalent of a.m. or p.m., whichever is appropriate
  cout << "time = " << time.to_string( "%I:%M:%S %p" ) << endl;

  // use default_format to set format, where
  // %H -- two-digit hour (00 to 23)
  // %q - microseconds (000000 to 999999)
  os_time::default_format( "%H:%M:%S:%q" ); // Change format.
  cout << "time = " << time << endl;


  // construct a date
  os_date date( 2, 12, 1965 );

  // use to_string to set format, where
  // %B -- month, using locale's full month names
  // %d -- two-digit day of month (01 to 31)
  // %y - year within century (00 to 99)
  // %Z - time zone abbreviation
  cout << "date = " << date.to_string( "%B %d %y (%Z)" ) << endl;

  // use default_format to set format, where
  // %c - date and time as %x%X
  // %x - date, using local's date format
  // %X - time, using local's time format
  os_date::default_format( "%c" ); // Change format.
  cout << "date = " << date << endl;

  return 0;
  }
```

List 3. Display Time and date using desired format

Output:

```
time = 11:04:22 AM
time = 11:04:22:000000
date = February 12 65 (CST6)
date = 02/12/1965 12:00:00 AM
```

Time Toolkit also contains classes for handling exceptions. It uses the ANSI/ISO C++ exception mechanism to report errors. Time Toolkit has a specialized error class that it uses to encapsulate the problem details. os_time_toolkit_error is an exception class for reporting errors within Time Toolkit.

## Conclusion

With its portability, simplicity, and easy to use APIs, Recursion's Time Toolkit is a very useful tool that all application programmers serious about time and date programming should have. It is a cost effective solution for time and date representation and management.

### Additional Resources

Time Zones in US. http://www.timetemperature.com/tzus/time_zone.shtml

Daylight Savings Time. http://www.spectrum-research.com/V2/daylight-savings-time.asp

Map of Time Zones. http://www.nationalatlas.gov/printable/timezones.html

Movement of Clock. http://www.timeanddate.com/worldclock/clockchange.html?n=263

For additional technical information on Recursion Software's
products and programs or for information on how to order and evaluate
Recursion Software technology, contact us today!

Recursion Software, Inc.
2591 North Dallas Parkway
Suite 200
Frisco, Texas 75034
1.800.727.8674 or 972.731.8800
www.recursionsw.com