www.photocase.com

Exploring the Perl DateTime module

COUNTING OUT TIME

Because calendar rules are influenced by historical and political decisions, date manipulations are riddled with pitfalls. Perl's DateTime module knows all the tricks. **BY MICHAEL SCHILLI**

If a backup script launches at 10 pm and quits at 4 am, how long did it take to run? Six hours? Well, it depends. Just think about a process that ran between March 26 and 27 in 2005 somewhere in the UK. The clocks were put forward by one hour at 1 am, and that would make five hours the right answer. If the same process had run at the same time in the USA, the answer would have been six hours, as summer time starts a week later in the US. But not in Indiana, which had not yet introduced summer time in 2005. In fact, Indiana is introducing summer time this year (2006, [2]). Fortunately, the DateTime module ([5]) from CPAN knows all these historical and future rules and provides an easy interface to even the most complex date calculations.

What if you wanted to know how long the current summer time rules have been in use in the UK? Listing 1 (*dsthist*) discovers this by looking back from the year 2006 and checking through the month of March to find out if there is a day where you end up at 5 a.m. when adding three hours and one second to 00:59:59. If this happens, summer time was used during this year, and the script

Listing 1: dsthist

```
#!/usr/bin/perl -w
use strict;
use DateTime;

YEAR:
for my $year (
 reverse 1964 .. 2006) {

 for my $day (1 .. 31) {

  my $dt = DateTime->new(
   year      => $year,
   month     => 3,
   day       => $day,
   hour      => 0,
   minute    => 59,
   second    => 59,
   time_zone =>
     "Europe/London",
  );

  $dt->add(
   hours   => 3,
   seconds => 1
  );

  if ($dt->hour() == 5) {
   print "$year: DST\n";
   next YEAR;
  }
 }
 print "$year: No DST\n";
 last;
}
```

stops when it discovers that's not the case. The display shows that 1972 was the first year with today's summertime rules:

```
...
1974: DST
1973: DST
1972: DST
1971: No DST
```

Summer in the City

Europe has fairly uniform daylight saving time rules, but this is not true of the American continent. This not only applies to the various countries; even some US states do their own thing, and there are even a few counties that do not adopt the same approach as the states in which they are located. And to make things even more complicated, the rules have changed in the course of time.

Listing 2 (*dstchk*) uses *all_names()* to ascertain all the timezones known to the DateTime::TimeZone module (which is also available from CPAN). It jumps to the first of January in the timezone it is investigating and adds six months. If this returns a date with an hour value that is not equal to zero, some kind of time adjustment must have occurred in the first six months of the year, meaning that this timezone must have switched to summer time in this period.

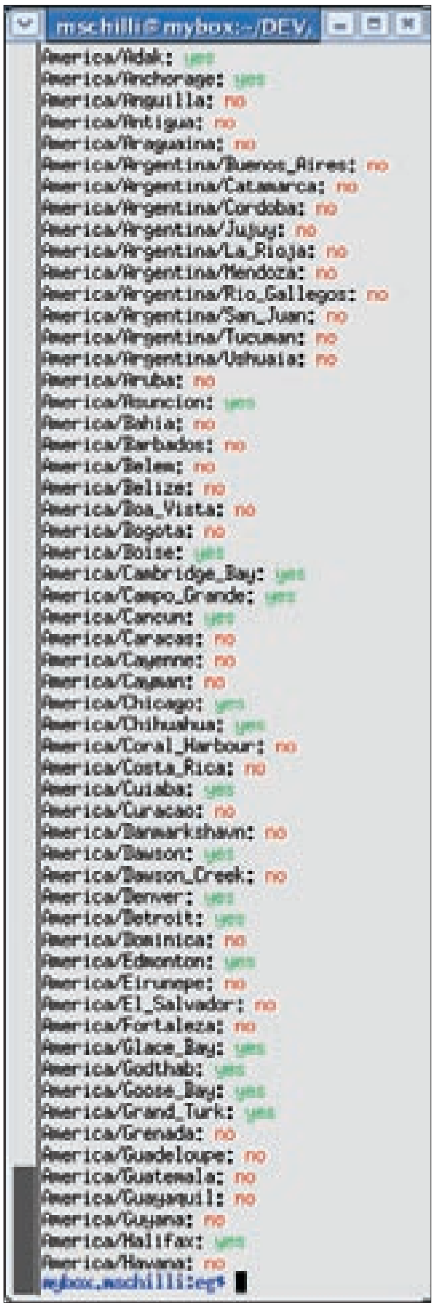
Zones are normally stored in a "*Continent/City*" format; examples are *Europe/London* (for Great Britain), *Europe/Dublin* (for Ireland), *America/New_York* (the state of New York in the USA), *America/Vancouver* (the Canadian state of British Columbia), and *Pacific/Honolulu* (for Hawaii). But if a county has deviated from state rules at some time in the past, an additional subdivision becomes necessary, for example, *America/Kentucky/Louisville* designates the US state of Kentucky, with its biggest city Louisville. (As you may be aware, Frankfort is the capital of Kentucky, although it is by no means the biggest town in the state.)

To reflect the fact that the county of Monticello in Kentucky was once (up to the year 2000) part of a timezone that is different from the one it is in now, *DateTime::TimeZone* has an entry for *America/Kentucky/Monticello*. Figure 1 shows the output from *dstchk*, showing daylight saving timezones in green using *Term::ANSIColor*.

DateTime can manipulate data for any timezone you like, including dates in the past, and timezones that have been through changes. Lord Howe Island just off the Australian coast has a quirky daylight savings time rule that puts the clock just half an hour forward or back. Listing 3 (*lord_howe*) shows that adding a second to 2005-10-30 01:59:59 returns a local time of 02:30:00.

A *DateTime* object created by the *new* constructor first exists in the special *floating* timezone, if the *time_zone* parameter doesn't specify the timezone explicitly. In this state, the timezone temporarily adapts to match other *DateTime* objects when calculations or comparisons are performed with them.

If you want to ignore daylight saving time in your time calculations, you can either use the "*floating*" zone, or you can choose the daylight-saving-time-free *UTC* (Universal Time Coordinated) timezone. Setting *time_zone* = > "*local*" for a DateTime object, to set the time to the zone in which your computer resides, forces *DateTime* to try out all kinds of tricks to guess your timezone configuration:



```
America/Adak: yes
America/Anchorage: yes
America/Anguilla: no
America/Antigua: no
America/Araguaina: no
America/Argentina/Buenos_Aires: no
America/Argentina/Catamarca: no
America/Argentina/Cordoba: no
America/Argentina/Jujuy: no
America/Argentina/La_Rioja: no
America/Argentina/Mendoza: no
America/Argentina/Rio_Gallegos: no
America/Argentina/San_Juan: no
America/Argentina/Tucuman: no
America/Argentina/Ushuaia: no
America/Aruba: no
America/Asuncion: yes
America/Bahia: no
America/Barbados: no
America/Belem: no
America/Belize: no
America/Boa_Vista: no
America/Bogota: no
America/Boise: yes
America/Cambridge_Bay: yes
America/Campo_Grande: yes
America/Cancun: yes
America/Caracas: no
America/Cayenne: no
America/Cayman: no
America/Chicago: yes
America/Chihuahua: yes
America/Coral_Harbour: no
America/Costa_Rica: no
America/Cuiaba: yes
America/Curacao: no
America/Danmarkshavn: no
America/Dawson: yes
America/Dawson_Creek: no
America/Denver: yes
America/Detroit: yes
America/Dominica: no
America/Edmonton: yes
America/Eirunepe: no
America/El_Salvador: no
America/Fortaleza: no
America/Glace_Bay: yes
America/Godthab: yes
America/Goose_Bay: yes
America/Grand_Turk: yes
America/Grenada: no
America/Guadeloupe: no
America/Guatemala: no
America/Guayaquil: no
America/Guyana: no
America/Halifax: yes
America/Havana: no
mybox.mschilli:eg*
```

Figure 1: Have you ever wanted a quick report on which areas of the American continent use daylight saving time?

Listing 2: dstchk

```
#!/usr/bin/perl -w
use strict;
use DateTime;
use Term::ANSIColor
  qw(:constants);

for my $zone (DateTime::
  TimeZone::all_names()) {

  my $from =
    DateTime->now(
   time_zone => $zone);

  $from->truncate(
   to => "year");
  my $to =
    $from->clone()
    ->add(months => 6);

  print "$zone: ";

  if ($to->hour() == 0) {
   print RED, "no", RESET,
     "\n";
  } else {
   print GREEN, "yes", RESET,
     "\n";
  }
}
```

Listing 3: lord_howe

```
#!/usr/bin/perl -w
use strict;
use DateTime;

my $dt = DateTime->new(
 year      => 2005,
 month     => 10,
 day       => 30,
 hour      => 1,
 minute    => 59,
 second    => 59,
 time_zone =>
   'Australia/Lord_Howe',
);

$dt->add(
 DateTime::Duration->new(
  seconds => 1
  )
);

# 2005-10-30 02:30:00
print $dt->date(), " ",
  $dt->hms(), "\n";
```

```
my $dt = DateTime->now(
   time_zone => "local");
print $dt->time_zone()->name();
```

This returned *America/Los_Angeles* on a machine located in our Perlmeister lab in San Francisco. Not bad.

Rotational Drag

How many seconds elapsed between 00:59:00 and 01:00:00 January 1 1999? One minute, that is 60 seconds? Wrong! Tip: At this point in time, the standard timezone, UTC, which is identical with the Greenwich timezone, added an extra *leap* second, which lead to a 61-second minute!

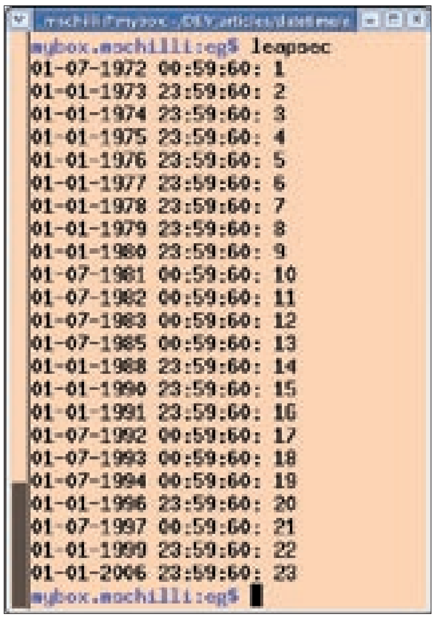
As the site at [3] tells you, a second is no longer defined as a fraction of one day, and hasn't been since 1967, but as a function of the far more constant resonance of the caesium 133 atom. The rotational speed of the earth has slowly decreased over the last 40 years. As the earth now takes slightly longer than 24 times 3600 atomic seconds for a single rotation, a leap second has been added here and there to official timekeeping since 1972, every 1.5 years on average. June 30 and December 31 are the reference dates. If official time is off by about one second, the second is added at the end of these days. This said, the earth seems to be back up to speed; 2005 was the first year since 1998 to need a one-second boost. If the earth were to rotate more quickly, the strict timekeepers would simply subtract a leap second from official time. However, this has never happened in the brief history of leap seconds.

Listing 4 (*leapsec*) starts in the year 1960 and moves gradually up to the year 2005, searching for leap seconds on June 30 or December 31. To do so, it sets the time in the UTC timeone to 23:59:00, adds 60 seconds, and checks if an unusual value of 60 is returned for the seconds segment. If so, the minute in question must have had 61 seconds, and we have discovered a leap second. After completing this search, the timezone is set to "Europe/London" by calling *set_time_zone()*. A call to *print* outputs the local time and the number of leap seconds found so far.

Figure 2 shows the output. The different local leap times on July 1 are a result of historical daylight saving time changes.

Pot-Holed Abstractions

Unfortunately, legacy timekeeping on Unix systems, which counts the seconds since 1970, does not take leap seconds into consideration. Whereas the second



```
mybox.mschilli:eg$ leapsec
01-07-1972 00:59:60: 1
01-01-1973 23:59:60: 2
01-01-1974 23:59:60: 3
01-01-1975 23:59:60: 4
01-01-1976 23:59:60: 5
01-01-1977 23:59:60: 6
01-01-1978 23:59:60: 7
01-01-1979 23:59:60: 8
01-01-1980 23:59:60: 9
01-07-1981 00:59:60: 10
01-07-1982 00:59:60: 11
01-07-1983 00:59:60: 12
01-07-1985 00:59:60: 13
01-01-1988 23:59:60: 14
01-01-1990 23:59:60: 15
01-01-1991 23:59:60: 16
01-07-1992 00:59:60: 17
01-07-1993 00:59:60: 18
01-07-1994 00:59:60: 19
01-01-1996 23:59:60: 20
01-07-1997 00:59:60: 21
01-01-1999 23:59:60: 22
01-01-2006 23:59:60: 23
mybox.mschilli:eg$
```

Figure 2: Leap seconds from 1960 to today.

Listing 4: leapsec

```
#!/usr/bin/perl -w
#################################
# leapsec - Print years with leap seconds
# Mike Schilli, 2005 (m@perlmeister.com)
#################################
use strict;
use DateTime;

my $secs;

for my $year (1960..2005) {
  for my $date ([30,6], [31,12]) {
     my $now = DateTime->new(
           year   => $year,
           month  => $date->[1],
           day    => $date->[0],
           hour   => 23,
           minute => 59,
           second =>  0,
           time_zone => "UTC");

     my $later = $now->clone()->add(
             seconds => 60);

     $later->set_time_zone("Europe/London");

     if($later->second() == 60) {
         print $later->dmy(), " ",
               $later->hms(), ": ",
               ++$secs, "\n";
     }
  }
}
```

Listing 5: leapreveal

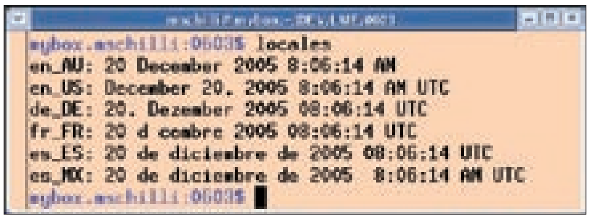
```
#!/usr/bin/perl -w
use strict;
use Sysadm::Install qw(:all);

use DateTime;

my $dt = DateTime->new(
 year      => 1990,
 time_zone => 'UTC'
);

$dt->add(
 seconds => 3600 * 24 *
   5000);
print "$dt\n";
```

hand moved from 23:59:59 to 23:59:60 December 31, 1998, in the UK, the counter on Unix machines that follow the POSIX standard moved from 915148799 to 915148800. The next virtual hop from 00:59:60 to 01:00:00, however, wasn't reflected by any Unix time counter, both points of time are correctly represented by a Unix time value of 915148800.

If you subtract two Unix times from one another, and calculate the UTC time that has elapsed between them, you may need to correct the results if a leap second has occurred between the two dates. For more details of this confusing approach, see [6] and [7].

DateTime provides the class method *from_epoch(epoch = > $time)*, which constructs a *DateTime* object from a Unix counter. The *epoch()* method of a *DateTime* object does the opposite, returning a counter value.

Listing 5 (*leapreveal*) shows what happens if you simply add the number of seconds in 5000 days to the date 1.1.1990: the result of this calculation is *2003-09-09T23:59:53*. In other words, the answer reveals that there are 7 seconds missing from the end of the day, caused by leap seconds in between the two dates! On the other hand, using *add(days = > 5000)* to add 5,000 days returns a result of *2003-09-10T00:00:00*. *DateTime* strictly separates the handling of time units such as days and seconds and will not normally convert a time value such as "5000 days" to seconds.



```
mybox.mschilli:0603$ locales
en_AU: 20 December 2005 8:06:14 AM
en_US: December 20, 2005 8:06:14 AM UTC
de_DE: 20. Dezember 2005 08:06:14 UTC
fr_FR: 20 d cembre 2005 08:06:14 UTC
es_ES: 20 de diciembre de 2005 08:06:14 UTC
es_MX: 20 de diciembre de 2005  8:06:14 AM UTC
mybox.mschilli:0603$
```

Figure 3: Different languages and customs applied to converting and formatting a date string.

But if you want to convert days to seconds, and will excuse the following potholed abstraction, you can use the *$to- > subtract_datetime_absolute($from)* method to subtract the DateTime object *$from* from a *DateTime* object *$to* to obtain a *DateTime::Duration* object, and this object's *seconds()* method really does give the exact number of seconds that elapsed during the period.

Superman

The dateline is another curiosity. If you fly east, the local time in the timezones you fly through gets later and later. At some point, the date has to shift to the previous day. If this were not so, you would be able to travel to the future in a fast plane. The dateline ([4]) crosses the Pacific from north to south, slightly to the east of the island groups off the South East Asia shore.

Listing 6 (*daytrip*) shows what happens if you take a fast plane from Japan (west of the date line) to Hawaii (east of the dateline). These are your flight details:

```
Departure:  Sunday, ↩
January 29 2006, 07:30
Arrival: Saturday, ↩
January 28 2006, 19:00
```

In other words, you leave on Sunday morning and arrive a day earlier, although the flight takes six-and-a-half hours. Saturday evening before the lottery results are announced – pity this only works for local time.

Speaking in Tongues

Listing 6 (*daytrip*) also shows how *DateTime* handles different date formats. It uses formating tools from the *DateTime::Format::** class hierarchy both to parse a date string with *parse_datetime()*, and to output the results. *DateTime::Format::Strptime* is a particularly flexible formatting tool that has placeholders for the format string, following a similar approach to the *strptime()* function in C. *%A* represents the weekday, *%B* the month name in writing, *%d* the date, *%H* the hour, and so on. The *locale* parameter is set to "*en_GB*" for Great Britain. *en* selects the English language. Part two of the locale specifies the country and its special rules.

Listing 6: daytrip

```
#!/usr/bin/perl -w
use strict;
use DateTime;
use
  DateTime::Format::Strptime;

my $format =
  DateTime::Format::Strptime
  ->new(
 pattern =>
   "%A, %B %d %Y, %H:%M",
 locale    => "en_UK",
 time_zone => 'Asia/Tokyo',
  );

my $dt =
  $format->parse_datetime(
"Sunday, January 29 2006,
   07:30"
  );

$dt->set_formatter($format);

print "Departure:  $dt\n";

$dt->add(
 DateTime::Duration->new(
  hours   => 6,
  minutes => 30
 )
);

$dt->set_time_zone(
 'Pacific/Honolulu');
print "Arrival: $dt\n";
```

Listing 7: locales

```
#!/usr/bin/perl -w
use strict;
use DateTime;
use
  DateTime::Format::Strptime;

my $dt = DateTime->now();

for my $locale (
 qw(en_AU en_US de_DE fr_FR
 es_ES es_MX)
  )
{

 $dt->set_locale($locale);

 my $format =
   DateTime::Format::Strptime
   ->new(
  pattern => $dt->locale()
    ->long_datetime_format()
   );

 $dt->set_formatter($format);
 print "$locale: $dt\n";
}
```

In Listing 7 (*locales*) are more examples: *en_AU* and *en_US* are locales for Australian and US English. *fr_FR* selects French; *es_ES*, and *es_MX* give you Spanish for Spain and Mexico. After initializing the formatter with an appropriate locale value, it is passed to the *DateTime* object using *set_formatter*. “Stringified” *DateTime* objects are converted to strings. Figure 3 shows a few examples.

Leap Years

Looking back to the year 2000, it is probably safe to assume that most programmers are aware of the rule that a leap year occurs every four years, but not if the year is divisible by 100, the exception being years that are divisible by 400.

Of course, *DateTime* understands these rules, so just to prove a point, let’s tackle a more complex problem: How long is the list of Friday the 29ths of February between 1980 and 2000?

Two sets of *DateTime* objects give us an elegant approach to solving this problem of finding the Friday leap days: we need to store all the Fridays in one of them, and all the 29ths of February in the other. A *DateTime::Set* class object can theoretically contain an infinite number of *DateTime* objects.

The easiest way to create a set for this purpose is to use the CPAN *DateTime::Event::Recurrence* module. The *DateTime::Event::Recurrence- > yearly(days = > 29, months = > 2);* constructor gives us a DateTime::Set type object with all the 29ths of February as an abstract description.

At the same time, Listing 8 (*frifeb29*) uses *weekly(days = > 5)* to define a second set that contains all Fridays (that is, all the 5th days in the week). We can then use the *intersection()* method to give us a set of Friday the 29ths.

To tell the iterator (defined in line 20) for the resulting set where to start, the *start* parameter defines the starting date, and a *DateTime* object set to the year 2020 is used to set the end date.

The *while* loop in line 29 uses *next()* for kicking off the iterator and pushing it towards the end of the period of time under review. The result of this investigation is: there was a Friday, February 29th in 1980, and there will be another one in 2008. ■

Listing 8: frifeb29

```
01 #!/usr/bin/perl -w
02 use strict;
03 use DateTime;
04 use
05   DateTime::Event::Recurrence;
06
07 my $feb29 =
08   DateTime::Event::Recurrence
09   ->yearly(
10  days   => 29,
11  months => 2
12   );
13 my $fri =
14   DateTime::Event::Recurrence
15   ->weekly(days => 5);
16
17 my $set =
18   $fri->intersection($feb29);
19
20 my $it = $set->iterator(
21  start => DateTime->new(
22   year => 1980
23  ),
24  end => DateTime->new(
25   year => 2020
26  ),
27 );
28
29 while (my $dt = $it->next())
30 {
31  $dt->set_locale("en_GB");
32  print $dt->day_name(), ", ",
33    $dt->month_name(), " ",
34    $dt->day(),        " ",
35    $dt->year(),       "\n";
36 }
```

INFO

[1] Listings for this article: *http://www.linux-magazine.com/Magazine/Downloads/64/Perl*

[2] “What time is it in Indiana?”, *http://www.mccsc.edu/time.html*

[3] Leap seconds: *http://en.wikipedia.org/wiki/Leap_second*

[4] The date line: *http://en.wikipedia.org/wiki/International_Date_Line*

[5] Datetime project homepage: *http://datetime.perl.org*

[6] Unix Time: *http://en.wikipedia.org/wiki/Unix_time*

[7] UTC, TAI, and UNIX time: *http://cr.yp.to/proto/utctai.html*

THE AUTHOR

Michael Schilli works as a Software Developer at Yahoo!, Sunnyvale, California. He wrote “Perl Power” for Addison-Wesley and can be contacted at *mschilli@perlmeister.com*. His homepage is at *http://perlmeister.com*.