Time Is The Simplest Thing

by Primoz Gabrijelcic

So said Clifford D Simak, but Simak got it all wrong. Time is indeed one of the most complicated things on this planet. We have made such a mess of it that it's hard even to agree on a meeting time with somebody living a few milliseconds away (in internet terms, that is).

It all started in 1961 (well, it all started maybe some ten billion years ago, but what's a few orders of magnitude between friends?) when Analog published *The Fisherman*, probably not the best novel Clifford D Simak had ever written. It's certainly readable, perhaps even entertaining, but I'm afraid not very memorable, except for a couple of sentences spoken by a friendly alien: 'Time, I'll tell you about time. Time is the simplest thing.' What an understatement.

Nowadays, time has a meaning only if you are discussing TV programmes with your significant other. Send an email to a friend and it may arrive before it was sent. Try to arrange an online meeting and you can be sure that only half of the invitees will be connected at the same time (and that won't be the arranged time). Save a file to a disk, then check it an hour later, and its creation time may well have changed.

There are too many times. There is Greenwich Time and Military Time and Pacific Standard Time and Co-ordinated Universal Time and Central European Time and Local Time and System Time and Swatch Internet Time and Too Late Time, to mention just a few of them. And it is a royal pain in the you-know-where to convert between them. When you think you have sorted it out, somebody (usually not your friend, at least not after his remark) reminds you of Daylight Saving Time and your beautiful solution just falls apart.

That happened to me, of course. I tried to write a simple routine that could convert a whole bunch of `TDateTimes` from local time to some more stable time base that doesn't make wild jumps twice a year. After two hours browsing MSDN and the internet, I was no closer to a solution. Apparently, Windows does not have this function, nobody on the internet has this function, and nobody cares about this function. Even more, I was hopelessly lost in the world of UTC, GMT, TZ, +0200, and similar stuff. I dare you: do you know what a leap second is? No? Well, neither did I.

Whilst browsing the net I did find a good collection of Delphi date and time functions: ESBDates. It contained a function that should have solved half of my problem. It was, however, desperately broken. Fear not, I reported that to the author and this broken function is not included in ESBDates anymore. Downgrading? Not at all, it is replaced with all the functions from the unit accompanying this article (which is on the companion disk of course). So if you're doing any date and time calculations, take a look at ESBDates at www.esbconsult.com.au since it will save you lots of work.

Greenwich Mean Time

In the beginning, there was chaos. Utter and complete chaos. Every town has its own 'sun' time. There was little communication between towns and therefore no pressure to establish a common time base. The first to feel this pressure was the navy when, in the fifteenth century, travel to the 'New World' flourished. To solve some problems, a common time base was established: Greenwich Mean Time (GMT). GMT is simply a time at some well-defined point, no more and no less than that. It is not affected by Daylight Saving Time (or Summer Time or whatever you may call it in your country).

Well, the British Navy was using GMT, but that was it. Continental people were just not interested in it. That changed only in the nineteenth century with the introduction of the railways. So, in 1880 Greenwich Mean Time became adopted officially by the British Parliament, in 1883 it was adopted by the USA, and in 1884 it was adopted internationally.

Today, GMT has lost some of its importance. It is defined in terms of Earth rotation, which is just not accurate enough. Therefore, in 1972 the new Coordinated Universal Time (UTC: don't ask me why they didn't call it Universal Time Coordinated to match the acronym!) was established. It is defined using the atomic clock and is always at the most 0.9 sec different to GMT. But GMT is still widely used, for example in email headers and internet cookies.

Actually, the terms UTC and GMT are often used interchangeably, as both standards define a time that is almost the same for all practical purposes. The typical personal computer clock is so unstable that you need to have some time synchronisation software installed if time is important to you and you can count on this time synchronisation software to keep you in sync with UTC. If you don't have such software installed, a good (and free) choice is Dimension 4: see the list of internet references at the end of the article for this URL and lots more.

Besides UTC there are also some less intelligent inventions like VRT (Virtual Reality Time) and Swatch Internet Time. Swatch time, for example, is a weird idea based on the concept that time zones are bad and it should be time 0 at the same moment everywhere.

I really could not find some 'hard' definition of Swatch Internet Time. All web documents contained mostly hand-waving: there is not much to base any conversion functions on. Defining Swatch Internet Time is not hard, it is

simply GMT plus one hour (without any Daylight Saving Time nonsense), where the time portion is specified as an integer in the range 0 to 999 (called Beat, usually marked with a prefix @). That is all. If you need a conversion function to convert between UTC and Swatch Internet Time, take it from Listing 1. UTC time is represented with a simple `TDateTime` variable, while Swatch Internet Time is stored in two variables, one holds the date part and other holds the Beat part. This complication is required because the date part of Swatch Internet Time is not always equal to the date part of Coordinated Universal Time. For example, the UTC time of 23:59 on 31st December 1999 corresponds to GMT+1 time 00:59 on 1st January 2000, which is the same as @041 on 1st January 2000 Swatch Internet Time.

Let's state this again: while local time changes as you traverse the globe, Swatch time does not. For example, when the local time is 16:42 in Central Europe, the local time in the United Kingdom is 15:42 and the local time in Canada is 8:42, but it is @654 all over the

TTimeZoneInformation Record And Day-In-A-Month Format

Delphi's `TTimeZoneInformation` record (which is defined in Windows.pas) is just an alias for Windows' TIME_ZONE_INFORMATION structure and specifies information specific to the time zone:

```
_TIME_ZONE_INFORMATION = record
  Bias: Longint;
  StandardName: array[0..31] of WCHAR;
  StandardDate: TSystemTime;
  StandardBias: Longint;
  DaylightName: array[0..31] of WCHAR;
  DaylightDate: TSystemTime;
  DaylightBias: Longint;
end;
TTimeZoneInformation = _TIME_ZONE_INFORMATION;
TIME_ZONE_INFORMATION = _TIME_ZONE_INFORMATION;
```

The `TTimeZoneInformation` fields are as follows:

Bias
This specifies the current bias, in minutes, for local time translation on this computer. The bias is the difference, in minutes, between Coordinated Universal Time (UTC) and local time. All translations between UTC and local time are based on the formula UTC equals local time plus bias.

StandardName
This specifies a null-terminated string associated with standard time on this operating system. This string is not used by the operating system, so anything stored there using the `SetTimeZoneInformation` function is returned unchanged by the `GetTimeZoneInformation` function. This string can be empty.

StandardDate
This specifies a `TSystemTime` record (an alias for Windows' SYSTEMTIME structure) that contains a date and local time when the transition from daylight saving time to standard time occurs on this operating system. If this date is not specified, the `wMonth` field of the record must be zero. If this date is specified, the `DaylightDate` field must also be specified. `StandardDate` field is formatted either as an absolute date or in a day-in-a-month format (see the notes later).

StandardBias
This specifies a bias value to be used during local time translations that occur during standard time. This field is ignored if a value for the `StandardDate` field is not supplied. This value is added to the value of the `Bias` field to form the bias used during standard time. In most time zones, the value of this field is zero.

DaylightName
This specifies a null-terminated string associated with daylight saving time on this operating system. This string is not used by the operating system, so anything stored there by using the `SetTimeZoneInformation` function is returned unchanged by the `GetTimeZoneInformation` function. This string can be empty.

DaylightDate
This specifies a `TSystemTime` record that contains a date and local time when the transition from standard time to daylight saving time occurs on this operating system. If this date is not specified, the `wMonth` field of the record must be zero. If this date is specified, the `StandardDate` field must also be specified. `DaylightDate` field is formatted either as an absolute date or in a day-in-a-month format (see notes below).

DaylightBias
This specifies a bias value to be used during local time translations that occur during daylight saving time. This field is ignored if a value for the `DaylightDate` field is not supplied.

This value is added to the value of the `Bias` field to form the bias used during daylight saving time. In most time zones, the value of this field is - 60.

Absolute Date
To specify a transition date in absolute format, set `wDay`, `wMonth`, and `wYear` to the day, month and year when the transition should occur and leave the `wDayOfWeek` field empty. Fields `wHour`, `wMinute`, and `wSecond` specify the exact time when the transition should occur.

Day-In-A-Month Format
To specify a transition date in day-in-a-month format, set `wHour`, `wMinute` and `wSecond` as for absolute date. Set `wYear` field to zero, `wDayOfWeek` field to an appropriate weekday (0 = Sunday), and the `wDay` field to a value in the range 1 through 5. Values 1 to 4 specify the corresponding week while 5 means 'last'. For example, setting `wDayOfWeek = 2` and `wDay = 3` means the third Tuesday in a month, while `wDayOfWeek = 0` and `wDay = 5` means the last Sunday in a month.

world. This makes time synchronisation somewhat easier (just tell the client to meet at @512) but it also introduces all sorts of problems. You still have to know what the time difference between your areas and your client's part of the world is (you don't want to schedule the meeting at 2am, do you?) and converting from Swatch time to local time is not as easy as adding or subtracting the appropriate number of hours (the difference between your local time and your client's local time). All in all, it looks like Swatch time brings more problems than it takes away.

The Beautiful World Of Time Zones

Swatch time is a nice thing but it doesn't solve all of our problems. It would surely be interesting if all the places on this planet could have the same time. It would surely be interesting as some would have sun at high noon and some could enjoy a look at the stars. It would surely be interesting, but probably not very practical, as you can imagine.

Therefore, we have time zones: areas with equal time which are so narrow that the sunrise time does not differ very much from one end of the time zone to the other. Time zones started at the same time as GMT. The USA was split into four time zones in 1883 and the world was divided into 24 time zones in 1884. The 24 standard meridians, every 15° East and West of 0° at Greenwich in the United Kingdom, were designated as the centres of the zones. The international dateline was drawn to generally follow the 180° meridian in the Pacific Ocean. Because some countries, islands and states do not want to be divided into several zones, the zones' boundaries tend to wander considerably from straight north-south lines.

And then there are the troublemakers: time zones that are not whole numbers away from GMT. In Australia, for example, where there are two time zones which are both 9 hours 30 minutes away from GMT. The USA has a few states that are in one time zone when

```
const
  MINUTESPERDAY = 1440;
type
  TSwatchBeat = 0..999;
function UTCToSwatch(utctime: TDateTime; var internetDate: TDateTime):
  TSwatchBeat;
begin
  utctime := FixDT(utctime+60/MINUTESPERDAY);
  internetDate := Trunc(utctime);
  Result := Round(Frac(utctime)*(High(TSwatchBeat)+1));
end; { UTCToSwatch }
function SwatchToUTC(internetDate: TDateTime; internetBeats: TSwatchBeat):
  TDateTime;
begin
  Result :=
FixDT(Trunc(FixDT(internetDate))+(internetBeats/(High(TSwatchBeat)+1))-60/MINUTES
  PERDAY);
end; { SwatchToUTC }
```

➤ *Listing 1: Swatch Internet Time conversion.*

```
function UTCToLocalTime(utctime: TDateTime): TDateTime;
var
  TZ : TTimeZoneInformation;
begin
  GetTimeZoneInformation (TZ);
  Result := UTCToTZLocalTime(TZ,utctime);
end; { UTCToLocalTime }
```

➤ *Listing 2: Converting UTC to local time.*

Standard Time is active and in another when Daylight Saving Time takes over.

Even stranger situations appear. For example, because of the Olympic Games 2000, several Australian states changed their Daylight Savings start date from the last Sunday in October to the last Sunday in August, creating a mess in the computer world.

The Horror Of Daylight Saving Time

I have already mentioned Daylight Saving Time (DST for short) a few times in this article, but I never told you much more about it. I have a good reason for that: Daylight Saving Time really scares me. It is a British invention that twice a year completely messes up my internal clock. It was invented by William Willet in 1907. He proposed to improve our 'health and happiness' (it doesn't apply to me) by advancing the clocks by twenty minutes on each of four Sundays in April, and by reversing this idea by the same amount on four Sundays in September. He also reckoned that this would save the country some £2.5 million.

The Daylight Saving Bill was introduced in 1909 but met with no success before the First World War broke out. In 1916, Daylight Saving Time was introduced as a wartime measure of economy, not only in Britain but also in many other countries. Most countries abandoned DST after the war had finished, then went on to reintroduce it eventually.

Besides making my life a misery twice a year, DST also introduces problems into time calculations. Twice a year the clock makes a weird jump: on one occasion one hour of local time is lost, on the second occasion one hour repeats itself. The second problem related to time calculations is that the changeover time is usually specified in the form 'last Sunday of October', which is not very easy to implement in software. A third problem is that on New Year's Eve, Standard Time is active in the northern hemisphere and Daylight Saving Time in the southern hemisphere.

Now let's try to program some useful functions. First, we need to get information about Daylight Saving Time for the current time zone from the Windows (forget about other time zones, we will deal with them later). The API function `GetTimeZoneInformation` will do the trick. It takes one parameter of type `TTimeZoneInformation` and fills it with relevant data. `TTimeZoneInformation` is quite a

large record, which is described in the sidebar *TTimeZoneInformation Record And Day-In-A-Month Format*.

Basically, the `TTimeZoneInformation` record is divided into three parts. `Bias` specifies the current bias, in minutes, for local time translation on this computer. `StandardName`, `StandardDate` and `StandardBias` specify the Standard Time, and the corresponding `Daylight` members specify the Daylight Saving Time. The `Name` part is a time zone name. `Date` specifies the date and time when the transition occurs (more on that later) and `Bias` is the bias that is used when this time (either ST or DST) is active. The bias is the difference, in minutes, between Coordinated Universal Time (UTC) and local time. All translations between UTC and local time are based on the formula UTC equals local time plus bias.

As we will write a generic set of functions, capable of dealing with any time zone, we will first create a simple wrapper function that will just retrieve the current time zone parameters and call a more generic function. You can look up the implementation of `UTCToLocalTime` in Listing 2.

There is not much work done yet, but now we come to the more interesting part. Function `UTCToTZLocalTime` (shown in Listing 3) takes time zone information and UTC time and returns the local time. First it calls the helper function `GetTZDaylightSavingInfoForYear`, which converts the time zone information into a more usable form, changeover times for a given year. For year 2000 and the Central Europe time zone it would return 26 March 2000 at 02:00 (the start of Daylight Saving Time) and 29 October 2000 at 03:00 (the start of Standard Time). If the time zone doesn't use DST, `GetTZDaylightSavingInfoForYear` returns false.

In the latter case, processing done in `UTCToTZLocalTime` is really simple. It just converts the time zone bias from minutes to a fractional part (as times are specified in `TDateTime`) and subtracts it from the UTC time.

If life were always that simple, I would not write a complete time zone library. Unfortunately, we already know that most parts of this world use some form of Daylight Saving Time and we must deal with it correctly. Therefore, `UTCToTZLocalTime` first converts times returned from helper function from local time to UTC by adding appropriate bias. It then checks whether the time zone is in the northern or southern hemisphere and calls another helper function called `Convert`. It simply checks which bias applies and subtracts it from the input time. It uses another helper function `DateEQ` to compare two times for equality because we cannot use a simple '=' to do this. `TDateTime` is stored as a real value and you cannot reliably compare two real values for equality. Instead of a simple equality test we subtract the time values and check if the difference is less then one tenth of one millisecond. As the Delphi `TDateTime` conversion functions (`EncodeDate` and `DecodeDate`) are only exact to one millisecond, this threshold looks appropriate.

At the end I should mention that function `Date2Year` belongs to ESBDates and simply returns the year part of the parameter.

Actually, converting from UTC to local time is easy, as this is an unambiguous operation. Converting from local time to UTC is much more complicated and tricky. So let us now deal with the dirty part and only then we will discuss the important function `GetTZDaylightSavingInfoForYear`.

➤ *Listing 3: Converting UTC to any time zone.*

```
function DateEQ(date1, date2: TDateTime): boolean;
begin
  Result := (Abs(date1-date2) < 1/(10*MSecsPerDay));
end; { DateEQ }
function UTCToTZLocalTime(TZ: TTimeZoneInformation; utctime:
  TDateTime): TDateTime;
  function Convert(startDate, endDate: TDateTime; inBias,
    outBias: longint): TDateTime;
  begin
    if ((utctime > startDate) or DateEQ(utctime,startDate))
      and (utctime < endDate) then
      Result := utctime - inBias/MINUTESPERDAY
    else
      Result := utctime - outBias/MINUTESPERDAY;
  end; { Convert }
var
  stdUTC : TDateTime;
  dayUTC : TDateTime;
  stdBias: longint;
  dayBias: longint;
  stdDate: TDateTime;
  dayDate: TDateTime;
begin { UTCToTZLocalTime }
  if GetTZDaylightSavingInfoForYear(TZ, Date2Year(utctime),
    dayDate, stdDate, dayBias, stdBias) then begin
    dayUTC := dayDate + stdBias/MINUTESPERDAY;
    stdUTC := stdDate + dayBias/MINUTESPERDAY;
    if dayUTC < stdUTC then
      // northern hemisphere
      Result := Convert(dayUTC,stdUTC,dayBias,stdBias)
    else
      // southern hemisphere
      Result := Convert(stdUTC,dayUTC,stdBias,dayBias);
  end else
    // TZ does not use DST
    Result := utctime - TZ.bias/MINUTESPERDAY;
end; { UTCToTZLocalTime }
```

Digging The Dirt

As you can see from a quick look at Listing 4, the function `TZLocalTimeToUTC` is a mess. I should add in my defence that this is not because I'm such a lousy programmer (at least, I hope so), but because the whole concept of Daylight Saving Time is broken. Just think about it. In 2000, Daylight Saving Time in the Central Europe time zone became active on 26 March 2000 at 02:00 when time was moved one hour forward. Ponder this again: local times from 02:00:00 to 02:59:59 never existed! The conversion function must somehow indicate that. I decided to return 0 in that case.

Even more interesting things happen when time changes back to Standard. On 29 October 2000 at 03:00, Central Europe time jumped

back to 02:00. One hour from 02:00:00 to 02:59:59 is repeated and we cannot tell if the local time of 02:30 corresponds to the UTC time 01:30 (as it would if the local time was using Standard Time at that moment) or 00:30 (if the local time was using Daylight Saving Time at that juncture).

This problem cannot be solved in a conversion function alone, but requires the programmer's help. That's why `TZLocalTimeToUTC` (and `LocalTimeToUTC`, which is again a simple wrapper) takes additional parameter `preferDST` that tells the function whether it should return DST or Standard Time in ambiguous cases. This overlap happens when DST changes back to Standard Time in all the existing time zones. In theory, this could also happen when Standard Time changes to Daylight Saving Time, but in that case DST would not deserve the name but should have been called Daylight *Losing* Time. The conversion function, however, handles both cases correctly.

If you don't know whether you should convert ambiguous local time according to the Daylight or Standard settings, you should, err, drop the responsibility on the user. A small code fragment in Listing 5 shows how to check for invalid and ambiguous times.

The Last Sunday Of October

We still don't know how to solve one important task: calculate the changeover times for a given year. Microsoft's documentation specifies that those times can be stored in `TTimeZoneInformation` in two formats, an 'absolute date' or a 'day-in-a-month' format (see the sidebar *TTimeZoneInformation Record And Day-In-A-Month Format* for further details).

The absolute date format is not actually very useful as it can only specify dates in one specific year (at least, that is what the official Microsoft documentation states) and is in fact never used. All the existing time zones are specified using day-in-a-month format. Even more, Microsoft's own Time Zone Editor (TZEDIT, found for example on the Windows NT Resource Kit

```
function TZLocalTimeToUTC(TZ: TTimeZoneInformation; loctime:
  TDateTime; preferDST: boolean): TDateTime;
  function Convert(startDate, endDate, startOverl, endOverl: TDateTime;
    startInval, endInval: TDateTime; inBias, outBias, overlBias: longint):
    TDateTime;
  begin
    if ((locTime > startOverl) or DateEQ(locTime,startOverl)) and
      (locTime < endOverl) then
      Result := loctime + overlBias/MINUTESPERDAY
    else if ((locTime > startInval) or DateEQ(locTime,startInval)) and
      (locTime < endInval) then
      Result := 0
    else if ((locTime > startDate) or DateEQ(locTime,startDate)) and
      (locTime < endDate) then
      Result := loctime + inBias/MINUTESPERDAY
    else
      Result := loctime + outBias/MINUTESPERDAY;
  end; { Convert }
var
  dltBias : real;
  overBias, stdBias, dayBias : longint;
  stdDate, dayDate : TDateTime;
begin { TZLocalTimeToUTC }
  if GetTZDaylightSavingInfoForYear(TZ, Date2Year(loctime), dayDate, stdDate,
    dayBias, stdBias) then begin
    if preferDST then
      overBias := dayBias
    else
      overBias := stdBias;
    dltBias := (stdBias-dayBias)/MINUTESPERDAY;
    if dayDate < stdDate then begin // northern hemisphere
      if dayBias < stdBias then // overlap at stdDate
        Result := Convert(dayDate, stdDate, stdDate-dltBias, stdDate, dayDate,
          dayDate+dltBias, dayBias, stdBias, overBias)
      // overlap at dayDate - that actually never happens
      else
        Result := Convert(dayDate, stdDate, dayDate+dltBias, dayDate, stdDate,
          stdDate-dltBias, dayBias, stdBias, overBias);
    end else begin // southern hemisphere
      if dayBias < stdBias then // overlap at stdDate
        Result := Convert(stdDate, dayDate, stdDate-dltBias, stdDate, dayDate,
          dayDate+dltBias, stdBias, dayBias, overBias)
      // overlap at dayDate - that actually never happens
      else
        Result := Convert(stdDate, dayDate, dayDate+dltBias, dayDate, stdDate,
          stdDate-dltBias, stdBias, dayBias, overBias);
    end;
  end else
    // TZ does not use DST
    Result := loctime + TZ.bias/MINUTESPERDAY;
end; { TZLocalTimeToUTC }
function LocalTimeToUTC(loctime: TDateTime; preferDST: boolean): TDateTime;
var
  TZ: TTimeZoneInformation;
begin
  GetTimeZoneInformation (TZ);
  Result := TZLocalTimeToUTC(TZ,loctime,preferDST);
end; { LocalTimeToUTC }
```

➤ *Listing 4: Converting any time zone to UTC*

```
date := TZLocalTimeToUTC(TZ,localTime,false);
if date <> 0 then begin
  // valid time
  date2 := TZLocalTimeToUTC(TZ,localTime,true);
  if not DateEQ(date,date2) then begin
    //ambiguous conversion
  end else begin
    //exact conversion
  end;
else begin
   //invalid local time
end;
```

➤ *Listing 5: Detecting ambiguous and invalid cases.*

```
function GetTZDaylightSavingInfoForYear(TZ: TTimeZoneInformation; year: word;
  var DaylightDate, StandardDate: TDateTime; var DaylightBias, StandardBias:
  longint): boolean;
begin
  Result := false;
  if (TZ.DaylightDate.wMonth <> 0) and (TZ.StandardDate.wMonth <> 0) then begin
    DaylightDate := DSTDate2Date(TZ.DaylightDate,year);
    StandardDate := DSTDate2Date(TZ.StandardDate,year);
    DaylightBias := TZ.Bias+TZ.DaylightBias;
    StandardBias := TZ.Bias+TZ.StandardBias;
    Result := (DaylightDate <> 0) and (StandardDate <> 0);
  end;
end; { GetTZDaylightSavingInfoForYear }
```

➤ *Listing 6: Calculating changeover times for given time zone and year.*

```
function DSTDate2Date(dstDate: TSystemTime; year: word): TDateTime;
begin
  if dstDate.wMonth = 0 then
    Result := 0
  else if dstDate.wYear = 0 then begin
    Result := DayOfMonth2Date(year, dstDate.wMonth,dstDate.wDay,
      dstDate.wDayOfWeek+1{convert to Delphi Style}) +
      EncodeTime( dstDate.wHour, dstDate.wMinute,dstDate.wSecond,
      dstDate.wMilliseconds);
  end else if dstDate.wYear = year then
    Result := SystemTimeToDateTime(dstDate)
  else
    Result := 0;
end; { DSTDate2Date }
```

➤ *Listing 7: Converting time zone transition date to TDateTime.*

```
function DayOfMonth2Date(year,month,weekInMonth,dayInWeek: word): TDateTime;
var
  days: integer;
  day : integer;
begin
  if (weekInMonth >= 1) and (weekInMonth <= 4) then begin
    // get first day in month
    day := DayOfWeek(EncodeDate(year,month,1));
    // get first dayInWeek in month
    day := 1 + dayInWeek-day;
    if day <= 0 then
      Inc(day,7);
    // get weekInMonth-th dayInWeek in month
    day := day + 7*(weekInMonth-1);
    Result := EncodeDate(year,month,day);
  end else if weekInMonth = 5 then begin
    // last week, calculate from end of month
    days := DaysInMonth(EncodeDate(year,month,1));
    // get last day in month
    day  := DayOfWeek(EncodeDate(year,month,days));
    day  := days + (dayInWeek-day);
    if day > days then
      // get last dayInWeek in month
      Dec(day,7);
    Result := EncodeDate(year,month,day);
  end else
    Result := 0;
end; { DayOfMonth2Date }
```

➤ *Listing 8: Converting day-in-a-month format to TDateTime.*

Supplement 2) cannot work at all with the absolute date format.

Because there are no guidelines on how to apply the absolute date format for an arbitrary year, I decided to work only with the day-in-a-month format. Absolute dates are only partially supported, time zone functions will return correct changeover times for the year specified in the absolute date and an error for all other years.

Let's now focus on functions that really do something useful. The function `GetTZDaylightSavingInfoForYear` (which is shown in Listing 6) implements hardly more than some basic data checking, calls `DSTDate2Date` and adds an appropriate bias to the result. `DSTDate2Date` (see Listing 7) uses `SystemTimeToDateTime` (defined in `SysUtils`) to deal with absolute dates and calls a real workhorse, `DayOfMonth2Date` (in Listing 8), for the day-in-a-month case.

We can be sure that Windows knows how to convert a day-in-a-month formatted date into something more readable. The only problem is that no such function is included in the Windows API. Sadly, they forgot to export it and therefore we have to write our own: I've already mentioned it and it's called `DayOfMonth2Date`.

The approach is straightforward. If the week is represented by a number from 1 to 4, this function starts by calculating the first day in the month. For example, Iran Standard Time starts on the fourth Tuesday of September and the first day of September 2000 was a Friday. It then calculates the first occurrence of the day we are searching for (Tuesday in our example; the first Tuesday in September 2000 was the 5th) and from that it is very simple to get *n*th occurrence of that day (the fourth Tuesday of September 2000 was the 26th).

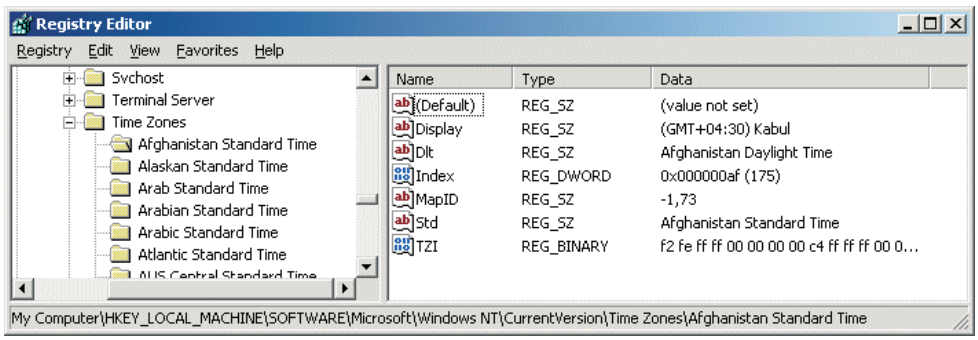
When week number 5 is specified (meaning the *last* week of the month, however many weeks the month may have), the approach is similar except that we have to work from the end of the month. First, we calculate the last day of the month with the help of the EDBDates function `DaysOfMonth`. For example, Ekaterinburg Daylight Time starts on the last Sunday of March and the last day of March 2000 was a Friday. From that, we calculate the last occurrence of the day we are searching for: the last Sunday in March 2000 was on the 26th.

A Time Zone That Is Not My Own

OK, but what if we want to convert a time from some other time zone? Where can we get the data? If you have the information about the changeover dates and biases, you can just put them into variables of type `TTimeZoneInformation` and call the appropriate function. If you don't have the changeover information, you can read it from the Windows registry.

It took me some time to find this information, but eventually I came across the Microsoft Knowledge Base article Q115231, which

➤ *Figure 1: Layout of time zone data in the Windows registry.*



```
type
  TRegTZI = packed record
    Bias: Longint;
    StandardBias: Longint;
    DaylightBias: Longint;
    StandardDate: TSystemTime;
    DaylightDate: TSystemTime;
  end;
function GetTZFromRegistry(reg: TBetterRegistry; var displayName: string;
  var TZ: TTimeZoneInformation): boolean;
var
  regTZI: TRegTZI;
begin
  Result := false;
  if assigned(reg) then begin
    with reg do begin
      // data in correct format - hope, hope
      if GetDataSize('TZI') = SizeOf(regTZI) then begin
        displayName := ReadString('Display');
        StringToWideChar(ReadString('Std'),@TZ.StandardName,
          SizeOf(TZ.StandardName) div SizeOf(WideChar));
        StringToWideChar(ReadString('Dlt'),@TZ.DaylightName,
          SizeOf(TZ.DaylightName) div SizeOf(WideChar));
        ReadBinaryData('TZI',regTZI,SizeOf(regTZI));
        TZ.Bias := regTZI.Bias;
        TZ.StandardBias := regTZI.StandardBias;
        TZ.DaylightBias := regTZI.DaylightBias;
        TZ.StandardDate := regTZI.StandardDate;
        TZ.DaylightDate := regTZI.DaylightDate;
        Result := true;
      end;
    end; //with
  end;
end; { GetTZFromRegistry }
```

➤ *Listing 9: Reading time zone data from the registry.*

describes the relevant registry key. All the data is stored in the key

```
HKEY_LOCAL_MACHINE\SOFTWARE\
  Microsoft\Windows NT\
  CurrentVersion\Time Zones
```

for Windows NT or 2000, or

```
HKEY_LOCAL_MACHINE\SOFTWARE\
  Microsoft\Windows\
  CurrentVersion\Time Zones
```

for Windows 95 and 98. Each time the zone is stored as a subkey (for example `Central Europe`) that holds five values: `Display` (the display name, which you can see in the Date/Time Control Panel applet), `Dlt` (Daylight Saving name), `Std` (Standard Time name), `MapID` (location on the map in the Date/Time applet) and `TZI` (time zone information). See Figure 1 for an example.

The data that is of interest to us is stored in the value named `TZI`. It is stored as a 44-byte record holding the biases (default, standard and daylight) and the changeover times. You can see it in Listing 9. While designing a set of functions to access this information I evaluated quite a few approaches but at the end decided to keep it simple. So I have created two functions. `GetTZCount` returns the number of time zones and `GetTZ` fetches all the relevant data from the registry and packs it into a `TTimeZoneInformation` record, which you can use as a parameter to other functions. All access to registry is done via the internal class `TBetterRegistry`, which is equal to `TRegistry` in Delphi 4 and higher, and implements the function `OpenKeyReadOnly` in Delphi 2 and 3.

Loose Ends

Not all the functions mentioned in this article are shown in print, as that would take much too much valuable space, so I recommend that you use the complete GpTimezone library on the disk. It's also on my website at

```
http://17slon.com/gp/gp/
  index.htm#GpTimezone
```

There you can also find the test program for GpTimezone (shown in Figure 2), which is almost a full-featured time zone editor, as well as updates. The source is, of course, included.

Finally, remember that all through this article we have been using the Gregorian calendar. If you want to complicate your life still further, read *Delphi For Time Travellers*, published in *The Delphi Magazine*, Issue 34 (June 98), for details of how to convert between different date systems.

Primoz Gabrijelcic is still pondering Simak's sentence. Nevertheless, he will try to respond to all mail sent to gabr@17slon.com. You may reuse the code that accompanies this article even if you have never read any of Clifford's novels.

➤ *Figure 2: GpTimezone test program.*