Date and time handling in Object Pascal

Michaël Van Canneyt

December 1, 2009

Abstract

Dates and times are encoded in various ways in different libraries and operating systems; The operating system may use different ways to encode a date or time depending on the use-case. An overview of how Object Pascal (as implemented in Delphi and Lazarus) handles dates and times.

1 Introduction

The representation of a date and a time) differs from API to API. The operating system has one (or more) views on date and time handling - different accross operating system, and each file format (in particular database formats) often also a particular way of encoding a date and time value - all of them try to minimalize the number of bytes to use in some way.

With the introduction of Object Pascal in Delphi, a `TDateTime` type was introduced, and used consistently throughout the RTL and VCL (in particular the database routines). It is declared in the System unit as follows:

```
Type
  TDateTime  = type Double;
```

Which means that it is equivalent to a Double, but is a new type nevertheless.

A `TDateTime` value is encoded a double-precision floating point value. It is encoded in accordance with the Microsoft Excel specification, also used in the Variant type of Microsoft's VB and COM technologies:

- The integer part (the part in front of the decimal point) of this floating point value contains the number of elapsed days since 30 Dec 1899. For more information on this topic, see also:

  ```
  http://support.microsoft.com/kb/214058/
  http://support.microsoft.com/kb/214330/
  ```

- The fractional part contains the number of elapsed milliseconds since the start of the day, divided by the total number of milliseconds in a day.

To demonstrate this, the output of the following small program is instructive:

```
program testd;

uses sysutils;
```

```
begin
  Writeln(DateToStr(0));
  Writeln(DateToStr(1));
  Writeln(DateToStr(35981));
end.
```

and looks like (`DateToStr` converts a date to a string representation):

```
12/30/1899
12/31/1899
07/05/1998
```

The exact output may differ slightly depending on the date settings of the computer, but the date it refers to should be the same. The first date may be encountered in many database programs; it is a sure sign that a programmer interpreted a NULL date value as zero somewhere in code, thus presenting the end-user with a rather meaningless date.

For time values, the following program

```
program testt;

uses sysutils;

begin
  Writeln(TimeToStr(0));
  Writeln(TimeToStr(1/2));
  Writeln(TimeToStr(0.99999998842));
end.
```

Will print (`TimeToStr` converts a date to a string representation):

```
00:00:00
12:00:00
23:59:59
```

Again, the actual output may differ depending on the time settings of the computer.

2 Encoding and decoding date and time

It is of course not necessary to calculate these floating point values manually. To create date and time values, or to decompose a Date/Time value, the SysUtils unit contains some functions:

```
Function EncodeDate(Year, Month, Day :word): TDateTime;
Function EncodeTime(Hour, Minute, Second,
                    MilliSecond:word): TDateTime;
```

The meaning of these functions is intuitively clear: They take a date (Y,MD) or time (H,M,S and Ms) specification, and encodes it according to the mechanism explained above.

If the values of these functions are not correct (when passing e.g. 42 for a day, or 77 for a number of minutes) then these functions will raise an exception. It is possible to avoid the exception by using the following functions:

```
function TryEncodeDate(Year, Month, Day: Word;
                       out Date: TDateTime): Boolean;
function TryEncodeTime(Hour, Min, Sec, MSec: Word;
                       out Time: TDateTime): Boolean;
```

These will return True or false depending on whether the arguments constitute a valid date/time, and return the encoded Date/Time in the last parameter.

If a full date/time value must be encoded, the following calls can be used from the `DateUtils` unit:

```
Function EncodeDateTime(const AYear, AMonth, ADay,
                        AHour, AMinute, ASecond,
                        AMilliSecond: Word): TDateTime;
Function TryEncodeDateTime(const AYear, AMonth, ADay,
                           AHour, AMinute, ASecond,
                           AMilliSecond: Word;
                           out AValue: TDateTime): Boolean;
```

The opposite operation - decomposing a date/time value into its various parts - exists as well in the SysUtils unit:

```
procedure DecodeDate(Date: TDateTime;
                     out Year, Month, Day: word);
procedure DecodeTime(Time: TDateTime;
                     out Hour, Minute, Second,
                         MilliSecond: word);
```

And likewise in the DateUtils unit:

```
Procedure DecodeDateTime(const AValue: TDateTime;
                         out AYear, AMonth, ADay,
                             AHour, AMinute, ASecond,
                             AMilliSecond: Word);
```

To Add a date and time part, the SysUtils unit contains the `ComposeDateTime` function:

```
Function ComposeDateTime(ADate,ATime : TDateTime) : TDateTime;
```

It is recommended to use this function: For dates after 30/12/1899, composing a date/time pair is just adding the date and time parts, but for dates prior to 30/12/1899, the composition is slighly more involved.

3 Windows date/time formats

Often, an operating system call returns a date/time value; This is either a date/time related to a file on disk, or the system time. The operating system APIs do not use the TDateTime format. The actual date/time format depends on the API call, and on the operating system. To be usable in most Object Pascal code, these values must be converted to TDateTime values.

On Windows, the `FindFirstFile` and `FindNextFile` calls use a structure `WIN32_FIND_DATA` which contains various information on file creation and access times; These times are noted in the `FILETIME` structure: A 64-bit number containing the number of 100-nanosecond intervals since 1/1/1601 00:00:00, in UTC.

This structure cannot be converted directly to a TDateTime value. This is a 3-Step process:

1. Converting the time to local (as opposed to UTC) time, using `FileTimeToLocalFileTime`, from the Windows unit.
2. Converting the local time to a DOS timestamp using `FileTimeToDosDateTime`, from the Windows unit.
3. Converting the DOS datetime to a `TDateTime` value using `FileDateToDateTime`, from the Sysutils unit.

The DOS DateTime format is discussed below.

The following function will return the last write time of a file as a `TDateTime` value:

```
Function FileTimeStamp(const AFileName: string): TDateTime;

var
  H : THandle;
  Info : TWin32FindData;
  LFT : TFileTime;
  DT : LongRec;

begin
  Result:=0;
  H:=FindFirstFile(PChar(AFileName),Info);
  if (H<>INVALID_HANDLE_VALUE) then
    begin
    Windows.FindClose(H);
    FileTimeToLocalFileTime(info.ftLastWriteTime, LFT);
    if FileTimeToDosDateTime(LFT,DT.Hi,DT.Lo) then
      Result:=FileDateToDateTime(Longint(DT));
    end;
end;
```

The 3 steps of the conversion are clearly visible in the last 3 statements.

The `GetLocalTime` function of the Windows API can be used to get the current local time of the system. This function takes a `TSystemTime` record as a parameter: it contains a field for each part of the date/time:

```
TSystemTime = record
  Year, Month, Day: word;
  Hour, Minute, Second, MilliSecond: word;
end ;
```

This makes it easy to convert to/from a `TDateTime` value. A function exists in the SysUtils unit which accomplishes this:

```
function SystemTimeToDateTime(const SystemTime: TSystemTime): TDateTime;
```

Its implementation is quite simple:

```
Function SystemTimeToDateTime(const ST: TSystemTime): TDateTime;
begin
  With ST do
```

```
    Result:=ComposeDateTime(DoEncodeDate(Year,Month,Day),
                            DoEncodeTime(Hour,Minute,Second,
                                         MilliSecond));
end;
```

4 Unix date/time formats

The unix API uses various date/time formats, again depending on the system call. For the filesystem, the various Stat calls (e.g. `fpStat`, `fpfStat` and `fpLStat`) return times in a `time_t` structure, which encodes a timestamp in a number of seconds since 1/1/1970, GMT. Converting this to a local `TDateTime` value is a 2 step process:

1. Convert the value to a local Date/Time using EpochToLocal from the Unix version of the SysUtils unit. This call returns the various date/time parts in separate variables:

   ```
   Procedure EpochToLocal(epoch:longint;var year,month,day,hour,minute,second:Wo
   ```

   It takes care of the timezones.

2. Encode the various parts obtained in step 1 to a TDateTime value using the various Encode functions explained above.

The `FileTimeStamp` function presented above would look as follows on unix:

```
Function FileTimeStamp(const AFileName: string): TDateTime;

Var
  Info : Stat;
  Y,M,D,hh,mm,ss : word;

begin
  If fpstat (pchar(FileName),Info)<0 then
    exit(0)
  else
    begin
    EpochToLocal(UnixAge,y,m,d,hh,mm,ss);
    Result:=ComposeDateTime(EncodeDate(y,m,d),
                            EncodeTime(hh,mm,ss,0));
end;
```

The `fpTime` system call returns the time of day, encoded using the same mechanism as the times in the stat record. It can therefor be converted in the same manner. The `fpgettimeofday` function also returns the time of day, but with a higher precision. It uses a `timeval` structure to return the time. This timeval structure has 2 fields:

```
Type
  timeval = record
   tv_sec: time_t;
   tv_usec: clong;
  end;
```

The first field tv_sec contains a timestamp with second precision, and the second part tv_usec contains the number of nanoseconds elapsed since the tv_sec second. Converting this to a TDateTime value uses the same mechanism (EpochToLocal) as above, and adds the number of milliseconds (obtained by dividing tv_usec by 1000).

All other date/time specifications in unix API calls are combinations of the above values.

5 DOS date/time formats

The DOS unit (in Free Pascal's RTL) also contains a FindFirst/FindNext set of calls, which return a record of type SearchRec with all available information about a file. The Time field of this record contains the time stamp of a file: The GetFTime call (also in the DOS unit) returns the same timestamp for an open file. The timestamp is encoded in the bits of this longint as follows:

Bits 0 - 4 Seconds divided by 2.

Bits 5 - 10 Minutes

Bits 11 - 15 Hours

Bits 16 - 20 Day of month

Bits 21 - 24 Month

Bits 25 - 31 Number of elapsed years since 1980.

These can be converted to a DateTime record using UnPackTime:

```
DateTime = packed record
  Year, Month, Day,
  Hour, Min, Sec: word;
End;
```

This record can be used to convert a dos timestamp to a TDateTime value using the EncodeDateTime function. The GetDate and GetTime calls return the various parts of the date and time in separate parameters, so they can also be easily converted to a TDateTime using the EncodeDateTime function.

6 SysUtils date/time formats

Although the SysUtils unit aims to be an abstraction of the Windows API, it still uses some Windows-specific types where file date and time are concerned: In a more consistent design, the TDateTime type would have been used consistently throughout its implementation.

The FindFirst/FindNext calls in the SysUtils unit are a platform independen version of the Windows FindFirstFile and FindNextFile calls. They use the TSearchRec record to return information about files. The Date field of this record contains the last modification time of a file, and this field is in DOS datetime format as defined by windows (not the DOS unit). As shown above, the DOS datetime can be converted to a TDateTime value using FileDateToDateTime, also declared in the Sysutils unit.

The FileAge function also returns a DOS DateTime value, and must therefor also be converted to a TDateTime value using FileDateToDateTime.

The `Date`, `Time` and `Now` functions return the current date, time or combined date/time as a `TDateTime` value.

7 Date/Time math

Till now, all routines were concerned with somehow obtaining a date and/or time value and converting this to a `TDateTime` format. None of this required manipulating date or time values. Manipulating dates in a TDateTime is easy: Since the date is encoded as a number of days, date math can be done by simply adding or substracting a desired number of days. However, not all operations are easily expressible using a number of days: for instance when one needs the same day in the next year, it's not sufficient to add 365 (this would go wrong in leap years). Therefor the DateUtils contains a number of functions to manipulate the date:

```
Function IncYear(D:TDateTime; YearCount: Integer = 1):TDateTime;
Function IncWeek(D:TDateTime; WeekCount: Integer = 1):TDateTime;
Function IncDay(D:TDateTime; DayCount: Integer = 1): TDateTime;
```

For historic reasons, the `IncMonth` function is in the SysUtils unit:

```
Function IncMonth(const AValue: TDateTime; const ANumberOfMonths: Integer = 1): T
```

Time calculations are slighly more complicated, because of the peculiar encoding of the time. These functions make it easier:

```
Function IncHour(T:TDateTime; HourCount: Int64 = 1):TDateTime;
Function IncMinute(T:TDateTime; MinCount: Int64 = 1):TDateTime;
Function IncSecond(T:TDateTime; SecCount: Int64 = 1): TDateTime;
Function IncMilliSecond(T:TDateTime;
                        MSecCount: Int64 = 1): TDateTime;
```

The number of days between 2 dates can easily be found by substracting the 2 dates from each other. In general finding the elapsed time between 2 dates (also counting partial years/months/days), the various `Span` functions can be used:

```
Function YearSpan(const ANow, AThen: TDateTime): Double;
Function MonthSpan(const ANow, AThen: TDateTime): Double;
Function WeekSpan(const ANow, AThen: TDateTime): Double;
Function DaySpan(const ANow, AThen: TDateTime): Double;
```

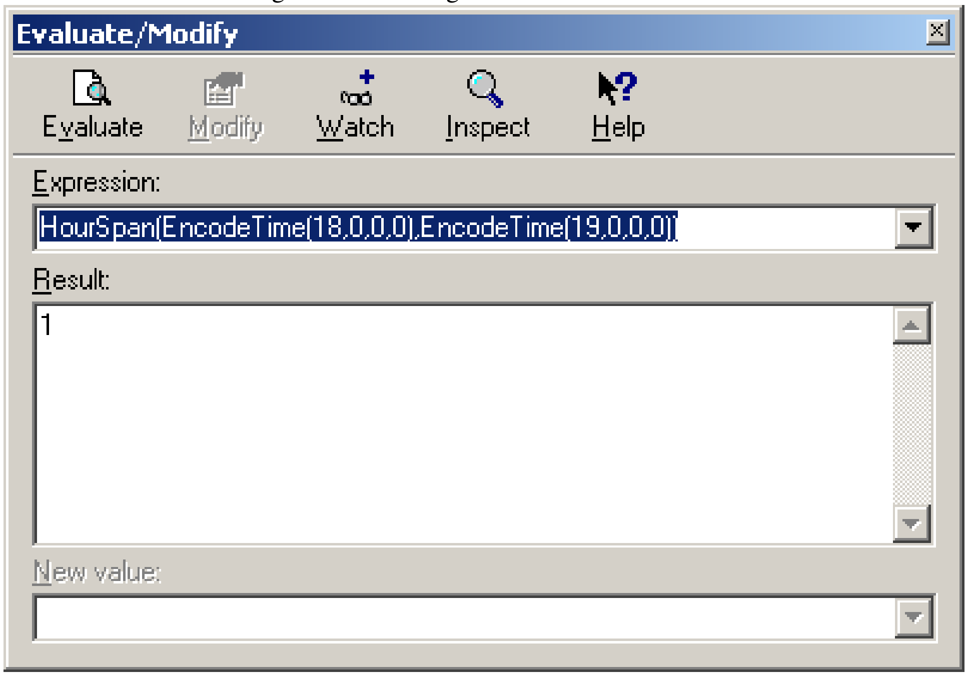
Although the first two will return an approximation, based on the average number of days in a month.

The following functions will return only return integer values, i.e. they truncate the result of the various `SPAN` functions, returning only periods that have completely elapsed:

```
Function YearsBetween(const ANow, AThen: TDateTime): Integer;
Function MonthsBetween(const ANow, AThen: TDateTime): Integer;
Function WeeksBetween(const ANow, AThen: TDateTime): Integer;
Function DaysBetween(const ANow, AThen: TDateTime): Integer;
```

Note that all these functions will operate correctly on TDateTime values that also have a time part. Similar functions exist for time values;

Figure 1: Rounding errors lead to confusion



```
HourSpan(EncodeTime(18,0,0,0),EncodeTime(19,0,0,0))
```

```
1
```

```
Function HourSpan(const ANow, AThen: TDateTime): Double;
Function MinuteSpan(const ANow, AThen: TDateTime): Double;
Function SecondSpan(const ANow, AThen: TDateTime): Double;
Function MillisecondSpan(const ANow, AThen: TDateTime): Double;
```

Note that these functions are not entirely accurate, because the time is stored as a fractional value. While the following

```
  D:=HourSpan(EncodeTime(18,0,0,0),EncodeTime(19,0,0,0));
  Writeln(D:5:2);
```

Will print '1.00', the following:

```
If HourSpan(EncodeTime(18,0,0,0),EncodeTime(19,0,0,0))>=1 then
  Writeln('One hour elapsed);
```

Will not print anything, because the HourSpan will return a value 0.9999999. In fact, the Delphi debugger's 'Evaluate' function will evaluate

```
HourSpan(EncodeTime(18,0,0,0),EncodeTime(19,0,0,0))
```

As '1', which is of course confusing (see figure 1 on page 8). One way to detect if an hour has elapsed (within second precision) is

```
If ((HourSpan(EncodeTime(18,0,0,0),
              EncodeTime(19,0,0,0)*3600)>=3599) then
  Writeln('One hour elapsed);
```

The DateUtils unit is devoted entirely to manipulating date/time values using the `TDateTime` type, and contains much more functions. a complete description is well past the scope of this article. The interested reader should consult the documentation of the unit.

8 String representations

Obviously, a user cannot read a TDateTime value of 0.5 as 12:00:00. This needs to be converted to a string representation of the time. Likewise, from a date value of 40148 it isn't exactly clear that 1/12/2009 is meant. The `DateToStr`, `TimeToStr` and `DateTimeToStr` transform a TDateTime value to a string representation which is human-readable. They are declared as follows:

```
Function DateToStr(Date: TDateTime): string;
function TimeToStr(Time: TDateTime): string;
function DateTimeToStr(DateTime: TDateTime): string;
function FormatDateTime(Fmt: String; DateTime: TDateTime): string;
```

Their output is determined from the following variables:

ShortDateFormat The format for short date notation, used in `DateToStr`

LongDateFormat The format for long (full) date notation.

ShortTimeFormat The format for short time notation.

LongTimeFormat The format for long time notation, in `TimeToStr` .

LongDateTimeFormat The format for long date/time notation, used in `DateTimeToStr`

The various `Format` strings consist of a combination of the values in 1. The same characters can be used in the `Fmt` argument to the `FormatDateTime` function.

The following variables are used when creating the string representations:

DateSeparator A character used as the separator character for dates (the character between the day and month or year parts).

TimeSeparator A character used as the separator character for times (the character between the hour and minute or second parts).

LongMonthNames An array with the full names of the months.

ShortMonthNames An array with the abbreviated names of the months.

LongDayNames An array with the full names of the days.

ShortDayNames An array with the abbreviated names of the days.

All these values are initialized from the operating system's internationalization settings. Note that in Free Pascal, for unix-like operating systems, the clocale unit must be included in the uses clause of the project, this unit will initialize the settings, but links to the C library to do this.

A common mistake is to assume that the following:

```
ShortDateFormat:='yyyy/m/d';
```

Will always result in a date like 2009/12/1. If the date separator char is '-', then the date will be printed as '2009-12-1', because the '/' sign is replaced by the actual date separator character. If the output must contain slash characters, the proper statement would be:

```
ShortDateFormat:='yyyy"/"m"/"d';
```

Table 1: Date/time formatting characters

Character	Meaning
c	Short date format (only for FormatDateTime)
d	day of month
dd	day of month (leading zero)
ddd	day of week (abbreviation)
dddd	day of week (full)
ddddd	Long date format (only for FormatDateTime)
m	month
mm	month (leading zero)
mmm	month (abbreviation)
mmmm	month (full)
y	year (two digits)
yy	year (two digits)
yyyy	year (four digits, with century)
h	hour
hh	hour (leading zero)
n	minute
nn	minute (leading zero)
s	second
ss	second (leading zero)
t	Short time format (only for FormatDateTime)
tt	Long time format (only for FormatDateTime)
am/pm	use 12 hour clock and display am and pm accordingly
a/p	use 12 hour clock and display a and p accordingly
/	insert date seperator
:	insert time seperator
"xx"	literal text
’xx’	literal text

Where the slash is enclosed in double quotes. The same is true for the timeformat. Unless surrounded by quotes, the ':' in the time format strings is replaced by the actual time separator character.

Note that the `FormatDateTime` function offers the most flexibility.

The inverse operation - converting a string to a TDateTime value - is also possible:

```
Function StrToDate(const S: ShortString): TDateTime;
function TryStrToDate(const S: ShortString;
                      out Value: TDateTime): Boolean;
Function StrToTime(const S: Shortstring): TDateTime;
function TryStrToTime(const S: AnsiString;
                      out Value: TDateTime): Boolean;
function StrToDateTime(const S: string): TDateTime;
function TryStrToDateTime(const S: AnsiString;
                          out Value: TDateTime): Boolean;
```

The functions without 'Try' in their name will raise an exception when the passed string does not contain a valid date/time. The functions with 'Try' will then simply return 'False'.

The meaning of these functions seems intuitively clear, but there is a caveat: the `StrToDate` function does not accept any kind of date. It will for instance not recognize '01 dec 2009', even if `ShortDateFormat` equals 'dd mmm yyyy'. It always assumes a date string using numerical notation: 'dd/mm/yyyy' or 'm/d/yy', and looks at `ShortDateFormat` only to determine the order of the day, month and year parts of the date. It does take the DateDelimiter character into account.

This has the peculiar effect that

```
  ShortDateFormat:='dd mmm yyy';
  StrToDate(DateToStr(Date));
```

Will raise an `EConverError` exception, saying that '1 dec 2009' is not a valid date.

The `ScanDateTime` function of Free Pascal (in the DateUtils unit) tries a little harder to detect date/time values in a string:

```
Function ScanDateTime(const Pattern:string;
                      const s:string;
                      startpos:integer=1) : tdatetime;
```

This will scan the string 'S', using `Pattern` as a format, starting at pos 1. `Pattern` contains the format to which the date conforms, using the same date formatting characters as indicated above. This function does work as expected:

```
uses sysutils,dateutils;

Var
  D : TdateTime;

begin
  ShortDateFormat:='dd mmm yy';
  D:=ScanDateTime(ShortDateFormat,DateToStr(Date));
  Writeln(DateToStr(D));
end.
```

will correctly print '01 dec 09' (if executed on 1 dec 2009).

9 Day of the week

When dealing with dates, the day of the week must sometimes be obtained. There are 2 functions that handle this. Both accept a TDateTime parameter with a date, and return the day of the week for thos date:

DayOfWeek in the SysUtils unit. It returns 1 for sunday, 2 for monday up to 7 for saturday - as used in many AngloSaxon countries.

DayOfTheWeek in the DateUtils unit. It returns 1 for monday, 2 for tuesday up to 7 for sunday.

The second one is ISO 8601 compliant, as are all routines in the DateUtils unit.

10 Conclusion

The `TDateTime` type is easy to work with, and the combined sysutils and `DateUtils` units contain enough routines to complete any task that one may need to do when dealing with dates and times, and this in a completely cross-platform manner, reducing the few caveats mentioned here to a very minor nuisance indeed.