Resolve That Date!

by Martin Lanza

In Issue 47, July 1999, Julian Bucknall detailed some neat date manipulation methods and routines in the regular *Algorithms Alfresco* column. I personally found this article both interesting and useful (as are most articles in *The Delphi Magazine [Give that man a Gold Star! Ed]*). There was, however, one area overlooked that disappointed me and this is probably a common issue with many Visual Basic developers who have migrated to Delphi in the quest for a better lot in life (sounds like a subject for *One Last Compile*?). The issue at hand is that Delphi is weak in the area of extracting and otherwise resolving dates from text format into the native `TDateTime` storage format. I recently found time to code a more flexible routine.

The problem at hand is that the Delphi `StrToDate` routine expects three numbers: month names are simply not accepted. On top of that the three numbers must be in the day, month and year order specified in the Windows date regional settings. The final blow is that only the date separator specified in the Windows date regional settings is accepted. These constraints make your average Visual Basic developer, who takes for granted throwing anything at the `DateValue` function, want to spit his false teeth out! Yes, I know, you can change many of the environment variables that control `StrToDate`, but why should you have to? In addition, this forces you to write more code, saving and restoring these settings, in case you upset other processing in your (and other) projects.

The goal is to have a simple and friendly routine that does not complain about what order the date components are in and is not particular about which date separator character has been used. I feel like I'm covering common ground here, many of you have most likely tackled this issue.

A bit of planning before we dive into a coding exercise. Resolving numeric values below 13 for dates is dangerous stuff: is it a day, a month, or a two-digit year? Resolving numeric values between 13 and 31 inclusive is only slightly less dangerous: is it a month or a two digit year? So here we are at the start of a new millennium; during the next thirty-one years we need to code more defensively than ever with regard to dates. I work primarily on treasury software and the consequences of making assumptions about two-digit years, and date formats in general, can be very serious, so I've decided to avoid making year assumptions and mandate that all years must have four digits. What can we do to reliably resolve the difference between a day and month? Most of the time when we are importing data from external sources (files, communications links, etc) the date format is known and constant. So all we need to

➤ *Listing 1*

```
unit Str2Date;
interface
function StrToDateNew(const sDateText: string;
  const bDayB4Month: boolean): TDateTime;
implementation
uses
  Windows, SysUtils;
function StrToDateNew(const sDateText: string; const
  bDayB4Month: boolean): TDateTime;
  { Try to convert a string into a Date. The year must be a
    FOUR digit year. Used PChars for performance, also step
    over all rubbish in in a single examination of the input
    string. Does not work on international MBCS :-( }
var
  sWord:  string;
  iValue: integer;
  bNumeric: boolean;
  PBeg, PNex: PChar;
  iInx, iDay, iMonth, iYear: integer;
const
  ZeroToNine = ['0'..'9'];
  Separators = ['|', '/', '\', '-', '_', ',', '.', ' '];
begin
  iDay    := 0;
  iMonth  := 0;
  iYear   := 0;
  // Begin at the start of the input date string.
  PBeg := PChar(sDateText);
  // Skip all leading blanks and separators
  while PBeg^ in Separators do
    Inc(PBeg);
  // Empty input (string) - Empty output (31-12-1899)
  // Could raise an exception instead - up to you.
  if PBeg^ = #0 then begin
    Result := 1;
    exit;
  end;
  // Initialize
  sWord := '';
  iValue := 0;
  PNex := PBeg;
  repeat
    // Have we got a number ?
    bNumeric := (PNex^ in ZeroToNine);
    if bNumeric then // Increment our integer value
      // 48 =        Ord('0')
      iValue := iValue * 10 + Ord(PNex^) - 48;
    Inc(PNex); // Step forward to the next character ...
    if (PNex^ = #0)  // End of String
      // Process our current item
      or (PNex^ in Separators) then begin
        if bNumeric then begin // Process iValue
          case iValue of
            1899..3000 :
              iYear := iValue;
            13..31 :
              iDay := iValue;
            1..12 :
              if iDay <> 0 then
                iMonth := iValue
              else if iMonth <> 0 then
                iDay := iValue
              else if bDayB4Month then
                iDay := iValue
              else
                iMonth := iValue;
          end; // Case iValue of
          iValue := 0; // Reset
        end else begin // Process sWord
          // If we don't already have the month see if this
          // word is a month name.
          if iMonth = 0 then begin
            SetString(sWord, PBeg, PNex - PBeg);
            sWord := UpperCase(sWord);
            if (PNex - PBeg) > 3 then
              SetLength(sWord, 3);
            iInx := Pos(sWord,
              'JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC');
            if iInx > 0 then
              iMonth := Succ(iInx div 3);
          end; // if iMonth = 0 then
          sWord := ''; // Reset
        end; // Process sWord
        // Skip any blanks and separators
        while PNex^ in Separators do Inc(PNex);
          // Set start of next word or value item
          PBeg := PNex;
      end; // Processed our current item
    until PBeg^ = #0;
    Result := EncodeDate(iYear, iMonth, iDay);
  end;
end.
```

know is whether the day appears prior to the month; this information will be supplied to our routine to avoid the guesswork.

Our new function is declared as:

```
function StrToDateNew(
  const sDateText: string;
  const bDayB4Month: boolean):
  TDateTime;
```

The first parameter is for the input date, in a text string format, and the second `Boolean` parameter tells us if the day will be encountered before the month. Within the function it uses `PChar` pointers to index into the input string (without changing it) and processes the string in a single pass. It basically parses the input string, storing text or numerics, until we change between text and numeric modes, at which point we evaluate our string or number.

The code is shown in Listing 1, which is on the disk of course, along with a small test kit. The test application has two deficiencies: it uses simple message dialogs to output the results (no good for non-interactive test runs) and it only performs positive tests (testing that various invalid text inputs do fail is advisable).

So how does the new routine perform? Well, it's quite happy with all of the following date inputs: '1900.1.2', '1900.2.1', '2/1/1900', '1/2/1900', '1900-Jan-2', '1900-2-Jan', '2 Jan 1900' and 'Jan 2 1900' (provided we pass the *day before month* parameter correctly!). As far as performance goes, in a direct comparison with `DateToString` we are 2.4 times faster (as measured on my tired little Celeron 300a); however, it is slower when resolving month names, but we had to leave a challenge for the enthusiastic and intelligent readers of *The Delphi Magazine!*

Martin Lanza (mlanza@zip.com.au) is an IT professional with 15 years of client server development experience within the banking and finance industry, primarily focused on treasury systems.