Delphi, Dates And The Year Two Thousand

by David Sutherland

This article consists of three sections. The first is a look at the general issues surrounding the year 2000. In the second I'll describe how Delphi and Inprise database products handle any issues. The third section looks at some date handling problems and how they are solved using Delphi.

The Millennium Bug

In June 1986 an article appeared in the magazine *COMPUTING SA*, written by a South African named Chris Anderson. The headline read: 'The Timebomb in Your IBM Mainframe System'. What he was referring to was the fact that date fields, as standard, only contained a two-digit year field, the century always defaulting to 19. Chris Anderson stated in the article that 'No terrorist organisation or disillusioned hacker could plant a more skilful, destructive or international booby-trap'.

At the time, very little interest was aroused by this article, except in IBM, who threatened legal action. IBM took a simple stance: 'This problem is fully understood by IBM's software engineers who anticipate no difficulty in programming around it ... Put simply, then, the position is as follows: those users still working a two-digit field can continue to do so, always bearing in mind the need to convert to a four-digit field by the year 2000'.

So then, what did the software industry do about it? Well, the vast majority of companies took the first bit of IBM's advice, but ignored the second vital part of it, and so twelve years later, we are reading Millennium Bug doom and gloom stories everywhere.

Software issues are only a part of the problem: many of the microprocessors in use today both in computers and embedded systems can only support the two-digit date field. In many ways, the issue of computers is only one of cost: simply test the hardware and replace if required. After all, in personal computing, the average machine is obsolete after six months (in technology terms anyway). Embedded systems pose further problems: what if the chip controlling the lift fails at 0.01am on 1 January 2000, or even worse the kidney dialysis equipment. But such discussions are not for this article, you will be relieved to know! No, what concerns us here is the impact on the applications that we as developers have produced.

In the United Kingdom, the CSSA (the UK trade association for software, IT services and information industries), the NCC and the UK Corporate Software User Association have come up with a draft document *Definition of Year 2000 Software Product and System Testing Best Practice*. This includes the following definition of Year 2000 conformity: 'Year 2000 Conformity shall mean that neither performance nor functionality is affected by dates prior to, during and after the year 2000'. It goes on to detail a number of general recommendations and specific tests based on the four rules defined by the British Standards Institute.

First, no value for the current date will cause any interruption in operation. Second, date-based functionality must behave consistently for dates prior to, during and after year 2000. Third, in all interfaces and data storage, the century in any date must be specified either explicitly or by unambiguous algorithms or interfacing rules. Lastly, year 2000 must be recognised as a leap year.

Having just read that set of rules, how many of us can state, hand on heart, that all our applications fully comply with them?

Inprise Products

Inprise have made available a large amount of material on the subject of the year 2000. There is an excellent table at www.inprise.com/devsupport/y2000/ which shows that Interbase, Paradox and dBase do not have a problem with the year 2000. Paradox and dBase will have a problem on 1 January 10,000 and Interbase on 12 December 5941, but I can't envisage that as a problem for any developer reading this article. Let a future generation of developers worry about that (sounds familiar that statement, doesn't it...).

There is, however, one important adjustment that you should make to ensure the proper display of dates in the desired four figure configuration. This is to adjust the short date format from `dd/mm/yy` to `dd/mm/yyyy`. This can be achieved in two ways.

Firstly, you can request the user to manually adjust the short date format. In a Windows 3.1 machine choose the `International` icon in

```
If Pos('yyyy',ShortDateFormat)=0 then
  ShortDateFormat:=Copy(ShortDateFormat,1,Pos('yy',ShortDateFormat))+'yy'+
    Copy(ShortDateFormat,Pos('yy',ShortDateFormat)+1,Length(ShortDateFormat));
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
Function Modify2000(const Value: TdateTime): TdateTime;
Var Year,Month,Day : Word;
Begin
  Result := Value;
  DecodeDate(Result,Year,Month,Day);
  If Year <1950 then Result := EnCodeDate(Year+100,Month,Day);
end;
```

the Control Panel and then click on the `Date Format Change` button and set the `Century` option to `long`. Windows 95, 98 and NT4 users should select the `Regional Settings` icon in the Control Panel, click on the `Date` tab and amend the `Short Date` format to `dd/mm/yyyy`.

An alternative solution, and in my opinion a better one in that it doesn't rely on end users' ability to find their way round the system, is to set the short date format in the initialisation of your application. The code shown in Listing 1 will set the format to four figure year settings, while respecting the underlying regional date setting.

This overrides the global setting in `SysUtils`, which is read from the registry and will ensure that the application consistently uses four-digit years, regardless of the Windows setting. It does not, however, change the `ShortDateFormat` in the registry and so if you make use of the `TDateTimePicker` control, this still shows the date using the setting in the registry. To solve this problem, you either make use of an alternative date edit control or use the following function:

```
DateTime_SetFormat(
  DateTimePicker.Handle,
  'dd/mm/yyyy');
```

Example code is for use in the United Kingdom. To make use of this function you need to add the `Commctrl` unit into your `uses` clause.

While it is desirable to force your applications' users to use four-digit year date entry, the Inprise website listed above does show a way of handling two-digit year entry, providing that your application only has to deal with a hundred year date entry range and you can afford to pre-define that range. The code in Listing 2 converts any date before 1950 (in Delphi terms any two- digit year less than 50) to a date 100 years later. For use with a data- aware control, the function can be used to convert a date typed in, using a `TTable.BeforePost` event handler.

```
Procedure Tform1.Table1BeforePost(DataSet : TdataSet);
Var Tempdate : TdateTime;
Begin
  Tempdate := Table1.fieldbyname('date_Field).asDateTime;
  Tempdate := modify2000(tempdate);
  Table1.fieldbyname('date_field').asdatetime := tempdate;
end;
```

➤ *Listing 3*

The function and procedure in Listing 3 can be used in all versions of Delphi. Delphi 4 came with a new variable to control the interpretation of two-digit years, the `TwoDigitYearCenturyWindow`.

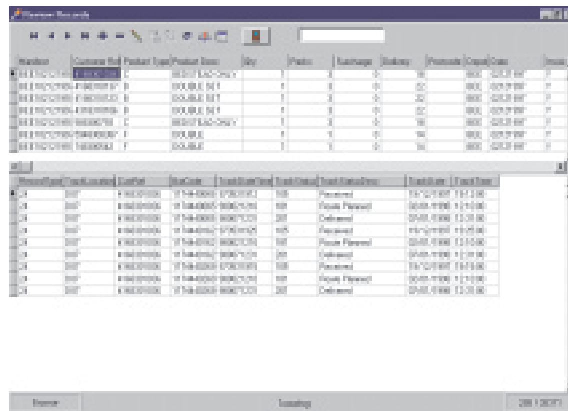
TwoDigitYearCenturyWindow

This is a global variable, which is used by `StrToDate` and `StrToDateTime` functions when converting a string to a date format. The value of this variable, if non-zero, is subtracted from the current date to form the pivot used to determine whether a year entered is in the current century or the next. Again you can only use this if your desired date range is 100 years or less. It replaces the `Modify2000 function` shown above. To best illustrate this, look at Table 1.

This makes the task in Delphi 4 of handling two-digit date entry easier, and has the advantage over the method required for early versions of Delphi that you only have to set one global variable. It is no real substitute for updating legacy applications to require four-digit year entry throughout, however.

➤ *Table 1*

Current Year	TwoDigitYearCenturyWindow	Pivot	yy = 03	yy = 50	yy = 68
1998	0 (default)	1900	1903	1950	1968
2002	0 (default)	2000	2003	2050	2068
1998	50	1948	2003	1950	1968
2000	50	1950	2003	1950	1968
2002	50	1952	2003	2050	1968
2020	50	1970	2003	2050	2068
2020	10	2010	2103	2050	2068



➤ *Figure 1*

Dates In Applications

Many of the applications that I develop import data from a number of sources. These often provide the date embedded within long strings and in a number of formats. An initial task, therefore, is to extract and manipulate these dates.

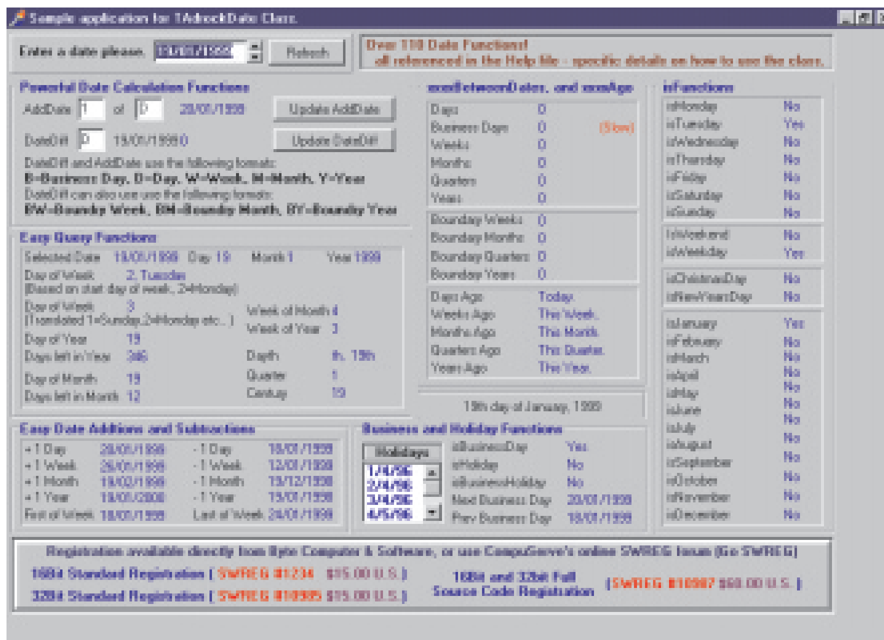
One application is a tracking and invoicing system. It takes header information about an order, for invoicing purposes, from one source in comma separated variable format and then adds tracking information about these orders. This information is received in fixed format text files.

The screenshot in Figure 1 shows the search screen, where you can see both the raw data and the extracted date and times.

The date information for the header panel is held within the manifest number, in the format `ddmmyyyy` and is extracted using the code in Listing 4.

This performs the task and as the raw data already had four-digit years needed no further work to make it year 2000 compliant.

The date information for the details was somewhat harder to deal with. The tracks are generated on a VAX VMS system and



➤ *Figure 2*

come in the format `yydddtt`, where `ddd` is the number of days that has elapsed since the beginning of the year. This was dealt with using the code in Listing 5.

You can see from this code that it uses an epoch setting to convert a two-digit year into a four digit year. However, it can only handle dates in a hundred year range between 1951 and 2050. It also uses the `Adrock` dates classes unit for the function `AddDays`. This function allows you to add a number of days to a date to obtain a new date. The main problem with constructing these procedures was the need to ensure that the data was of the right type at any one time.

Another important area of my work concerns the calculation of transit times, ie how long does a parcel take to get from A to B. This application again makes use of the `Adrock` date classes unit. In this case (see Listing 6), the function used is `ReturnBusinessDaysBetweenDates`, which calculates the number of business days that fall between the first date and the second date. If the second date is later than the first then the result will be negative. The business days are days that do not fall on a Saturday, Sunday or appear in the holiday list. The holiday list is a string list to which you can add or remove dates either at design-time or runtime. See www.adrock.com for more details of these routines.

This procedure first of all calculates the transit time for two separate parts of the order cycle. It then calculates what band the total transit time falls into, that is: under 5 days, between 5 and 10 days, between 10 and 15 days, between 15 and 20 days, or over 20 days.

➤ *Listing 4*

```
procedure TfrmInvoice.btnDespDateClick(Sender: TObject);
var
  Year, Month, Day, DespatchDate, Store : string;
  Save_Cursor:TCursor;
begin
  Save_Cursor := Screen.Cursor;
  Screen.Cursor := crHourglass;
  with tblHeaders do begin
    DisableControls;
    try
      First;
      while not EOF do begin
        { Process each record here. Splits the string into the relevant chunks }
        Day := Copy(tblHeadersDate2.AsString,1,2);
        Month := Copy(tblHeadersDate2.AsString,3,2);
        Year := Copy(tblHeadersDate2.AsString,5,4);
        Store := Copy(tblHeadersCustomerRef.AsString,1,3);
        tblHeaders.edit;
        {Combines the test strings into the appropriate format with separators}
        DespatchDate := Day +'/'+ Month + '/'+ Year;
        {converts string back to a date while casting datafield as a datetime}
        tblHeadersDespatchDate.AsDateTime := StrToDateTime(DespatchDate);
        tblHeadersStoreCode.AsString := Store ;
        tblHeaders.post;
        Next;
      end;
    finally
      EnableControls;
      tblHeaders.Active := True;
      Screen.Cursor := Save_Cursor; { Always restore to normal }
    end;
  end;
end;
```

➤ *Listing 5*

```
procedure TfrmUpdate.btnExtractClick(Sender: TObject);
var
  Year, Hours, Minutes, Days : string;
  DayInt, EpochSetting, YearInt: integer;
  Save_Cursor:TCursor;
begin
  Save_Cursor := Screen.Cursor;
  Screen.Cursor := crHourglass;    { Show hourglass cursor }
  // uses for manipulatin of two digit years
  EpochSetting := 50;
  with tblStatus do begin
    DisableControls;
    try
      First;
      while not EOF do begin
        { Process each record here }
        Year := Copy(tblStatusTrackDateTime.AsString,1,2);
        Days := Copy(tblStatusTrackDateTime.AsString,3,3);
        Hours:= Copy(tblStatusTrackDateTime.AsString,6,2);
        Minutes:=Copy(tblStatusTrackDateTime.AsString,8,2);
        DayInt := StrtoInt(Days) -1;
        YearInt := StrtoInt(Year);
        tblStatus.edit;
        if EpochSetting > YearInt  then
          Year := '1/1/19'+ Year
        else
          Year := '1/1/20' + Year;
        tblStatusTrackDate.AsDateTime :=
          StrToDateTime(DateToStr(AddDays(DayInt,
          StrToDateTime(Year)))+' '+Hours+':'+Minutes);
        tblStatusTrackTime.AsDateTime :=
          StrToTime(Hours + ':' + Minutes);
        tblStatus.post;
        Next;
      end;
    finally
      EnableControls;
      tblStatus.Active := True;
      Screen.Cursor := Save_Cursor;
    end;
  end;
end;
```

This pre-processing of the data to decide what band a particular transaction fell into was needed for reporting purposes. Calculating it at data entry time vastly simplified the task of producing the report. This was due to the difficulty of performing calculations such as totals, percentages and averages and so on upon data calculated at the time of running the report, as opposed to data available at design-time.

Figure 2 is a screenshot taken from the demonstration program included with the `TAdrockDateClass` which shows a number of the functions available in this product.

The problems encountered at first in designing the reports reminded me of the importance of ensuring that when you are under-taking the analysis work before starting to set up data structures and so on (you do this don't you!) you need to account for the desired output from the system, so that these requirements are taken into account at initial design-time before you start coding.

```
procedure TfrmData.tblTransitAfterEdit(DataSet: TDataset);
begin
  tblTransitTransit.Value :=
    ReturnBusinessDaysBetweenDates(tblTransitDelDate.AsDateTime,
      tblTransitRecDate.AsDateTime);
  tblTransitReceived.Value :=
    ReturnBusinessDaysBetweenDates(tblTransitRecDate.AsDateTime,
      tblTransitOrderDate.AsDateTime);
  tblTransitTotalTransit.Value :=
    ReturnBusinessDaysBetweenDates(tblTransitDelDate.AsDateTime,
      tblTransitOrderDate.AsDateTime);
  if tblTransitTransit.Value <= 5 then
    tblTransitDays5.Value := 1;
  if (tblTransitTransit.Value <= 10) and (tblTransitTransit.Value >5 ) then
    tblTransitDays10.Value := 1;
  if (tblTransitTransit.Value <= 15) and (tblTransitTransit.Value > 10) then
    tblTransitDays15.Value := 1;
  if (tblTransitTransit.Value <= 20) and (tblTransitTransit.Value > 15) then
    tblTransitDays20.Value := 1;
  if (tblTransitTransit.Value > 21) then
    tblTransitDays21.Value := 1;
end;
```

➤ *Listing 6*

Conclusions

I trust that this article has enabled you to gain an understanding of the issues behind the year 2000 and of the rules for determining whether an application can be said to be year 2000 compliant.

In addition, we've covered the importance of taking into account the problems faced when having to integrate data exported by other applications, where you normally have no control over the data structure employed, and the methods that you can employ to overcome such issues in the areas of dates and times.

David Sutherland is a professional information provider, developer and writer and can be contacted at davesuth@bigfoot.com

www.itecuk.com
News, contacts, what's coming, back issues, samples and more