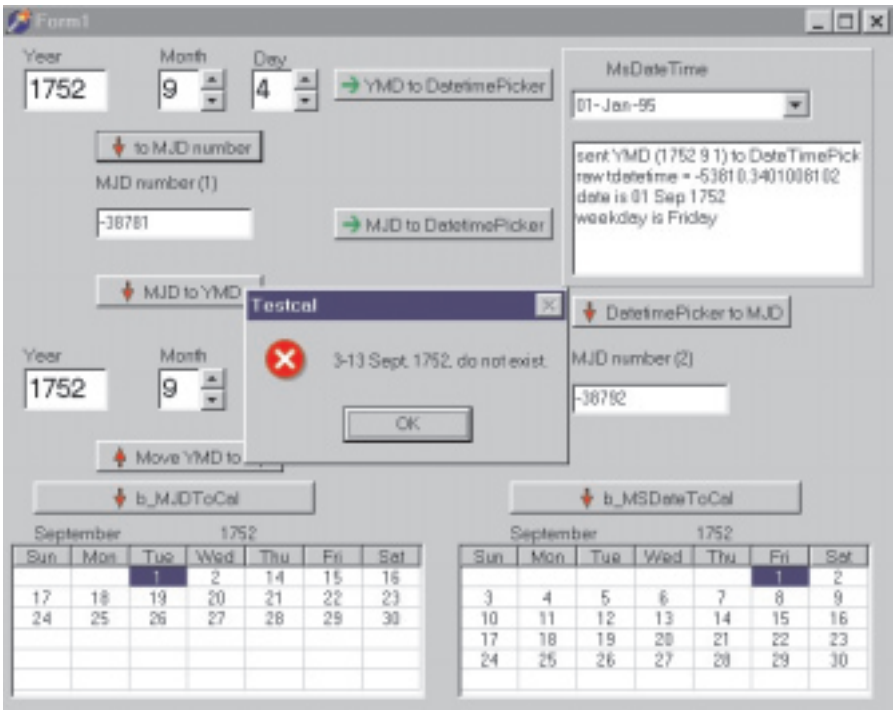
Delphi For Time Travellers

by Brandon Smith

If you are a Delphi 2 or 3 programmer, your universe starts on the 1st of January year 1 and will end on the 31st of December year 9999. This is not a problem for most of us, but for those doing the system software for a time machine, there are obvious drawbacks. Even within this ten thousand years Inprise has allowed our software to live, a time machine controller will run into some serious problems with `EncodeDate` and `DecodeDate`. This article discusses these problems and suggests techniques for dealing with date problems.

The first problem concerns time travellers going into the future and long-term planners. Under the current Gregorian system, a few seconds per year are not accounted for. Therefore, starting in the year 4000, the algorithm normally used to determine leap years won't work correctly. It'll put us off by a day approximately every 3323 years thereafter. The solution is simple: divide the year by 4000 and, if there is no remainder, it is not a leap year. However, even this solution is not perfect and will lead to the calendar being off by another day in about 100,000 years. The source for this disturbing news is the *Calendar* entry in the 1911 edition of the *Encyclopaedia Britannica*. I've no doubt the most recent edition contains the same facts, but the 1911 edition is what sits on my shelf.

Coming back from future concerns to the present, leap years are not nearly so much a problem as leap seconds. The Gregorian, Julian and Julian Day systems we'll be looking at operate on the assumption that the earth travels around the sun in a fixed period of time each year. However, it turns out this is no longer true, as is described in the boxout on problems not solved. While I don't think we need worry about accounting for leap seconds in most Delphi apps, these animals are of great importance to guided missile engineers and bass fishermen, since their Global Positioning System toys can only get back to that sweet spot on the lake if the time is accurate to 0.1 second per year.



➤ *Figure 1*

Of much greater import for Delphi programmers working on time travel control systems going into the past or appliations dealing with historical events or genealogy, is the fact that `EncodeDate(1752, 9, 12)` does not raise an exception. However, as I'll explain, 12th September 1752 never happened in the English speaking countries. Similarly, 10th October 1582 exists for the English speaking peoples, but does not exist for most of the parts of Europe which were Catholic at the time. It all starts back before Delphi/Microsoft dates exist, back in the days Before Christ.

When Julius Ceasar was bringing order to his empire, one of his major tasks was to align the timing of civil and religious events. Lest you think he was being altruistic, you should realize that before he tackled it, the temples had charge of the calendar. The high priests added or subtracted days so the religious celebrations would occur at auspicious astronomical times, but the high priests had also got into the habit of moving feast days around so that elections and other political events took place at times more advantageous to the temples than to the emperor. By the time Caesar turned his attention to it, the spring equinox festival, marking the start of the year, was taking place three months early, in the middle of winter. I won't confuse you or myself by trying to sort out how many months a year had before Caesar set up what has become known as the Julian calendar, nor will I delve into the strange way days were identified (which dismayed even Roman writers).

Julius delegated the task to his temporal tech support, a guy named Sosigenes. This expert came up with a year of 12 months to which a day was added every fourth year. The twelve months were to have alternating lengths of 30 and 31 days, except February, which would have 29 days for three

years, and 30 days on the fourth. To keep folks on their toes, though, the extra day wasn't added to the end. Instead it was inserted between the 24th and 25th of February, another factor which could make a time machine arrive on the wrong day, and another factor which we'll ignore in our code at this time. The Julian calendar is implemented by the following, which you may notice is not the logic used in Delphi:

```
function IsLeapYear(
  AYear: Integer): Boolean;
Begin
  Result := (AYear mod 4 = 0);
end;
```

After lengthening the current year, 47 BC, to 445 days to make up for how far off the reckoning had become, Caesar implemented the Julian calendar at the spring equinox, 25 March 46 BC (at least one source said it was 45 BC...). Unfortunately, when Sosigenes sent the formal change notification over to the users in the temples, they either misread his formula or he made a mistake and they implemented leap years every third year. This slight error was not noticed for 36 years and Augustus (Octavian) fixed it by issuing new user instructions stating that there would be no leap years for the next nine years.

Also during this early period, every committee in Rome got its hands into the pie, resulting in a month being renamed July in honor of Julius Caesar and, later, another month being renamed August in honor of Augustus. Since Augustus had to be at least equal to Caesar, August was also given 31 days. This required the number of days in the other months to be shifted around by various committees and emperors until we ended up with the arrangement we now have. Sometime during this period the beginning of the civil year was moved from March back to January.

There seems to be a consensus that the calendar structure as we know it has been reasonably consistent since about 8 AD. The numbering of the years and use of AD and BC didn't start until the 6th century. Was there a zero year? Let me quote from Dr Stockton's web page (see web sources boxout): 'The monk Dionysius Exiguus (Denis the Little), around what later became 523 AD, did his calculations, no doubt, in Roman numerals, so I imagine that he failed to consider the possibility of a year zero, and went ... II BC, I BC, I AD, II AD... Hence, on the present calendar, no year is numbered zero, and 1 AD directly follows 1 BC.'

If you happen across a reference to the year zero, you are probably reading an astronomical work, and that year is most likely the one before 1 AD.

To make sure we remain confused, 'Julian Date' is an entirely different system of counting days which was invented by Joseph Justus Scaliger in 1583. Most sources say he named his system after his father, Julius Caesar Scaliger, but who knows? In any case, Scaliger Cycle 1 started on 1 January 4712 BC (or 4713 BC) and will last 7980 (Julian) years, a period of time established by multiplying the solar cycle (28 years) by the lunar cycle (19 years) by the Indiction cycle (the 15 year schedule upon which ancient Roman taxes were levied).

Something in Scaliger's approach appealed to astronomers, and the Julian Day system is the standard for expressing astronomical events today, when they aren't using the Modified Julian

Julian Day Numbering

Refer to a good encyclopaedia for details on the life of Joseph Justus Scaliger and how he came up with the system he devised. What his Julian Day Numbering system does is establish a universally agreed upon way of establishing the time of astronomical events. The fact that various sources give different dates for the calendar date that corresponds to Julian Day One is not of any major concern since there seems to be nearly universal agreement that Julian Day Number (JDN) 2,400,000 is noon, GMT, 16 November 1858.

This makes JDN 0 equivalent to 1 January 4713 BC, or 1 January 4712 BC depending on whether or not you count the year before 1 AD as zero. You might also see JDN 0 identified as 25 November 4714 BC, in which case what's happening is that the Gregorian calendar system has been projected backwards in time past the point where it became official. This is called the Proleptic Gregorian calendar.

The reason the days start at noon instead of midnight is that astronomers work at night. Until 1 January 1925, GMT days started at noon.

Modified Julian Day, MJD, is JD minus 2,400,000.5. This makes MJD days start at midnight rather than noon and shifts the zero day to 17 November 1858. The main benefit is that modern astronomers have only to deal with 5 digits for most of their dates instead of the 7 they would need for regular Julian days. 10 October 1995 is MJD 50,000. MJD numbers have the additional property that (MJD + 999,990) MOD 7 yields the correct day of the week.

Truncated Modified Julian Day sometimes used by NASA is simply the MJD with the first digit chopped off, or, from another source I read, Truncated Julian Date is the number of days since the first Apollo mission, 1968-05-24, or some other date convenient for coding purposes.

You will also find other date numbering systems which are called Julian dates. When I was a clerk working out of a foxhole in Vietnam, I had to date supply requisitions with what they called Julian dates. These were simply the cardinal value of the day of the year preceded by the last digit of the year. 7020 was January 20th, 1967. You may also see Julian dates where 20 January 1967 would be 67020 or 1967020. So for Julian Dates, be aware of the source.

If your mystery date is in an official document published by the International Astronomical Union, the Consultative Committee for Radio, or the International Telecommunications Union, then you can be reasonably confident that you are looking at an MJD, and can proceed accordingly.

```
function IsLeapYear(AYear: Integer): Boolean;
begin
  Result := (AYear mod 4 = 0) and ((AYear mod 100 <> 0) or (AYear mod 400 = 0));
end;
```

➤ *Listing 1*

Day system. See the boxout on Julian Day Numbering.

As it happens, the time it takes the earth to move from one spring equinox to the next is 365 days, 5 hours, 48 minutes and 46 seconds of mean solar time, not the simple 365 and one quarter days that Sosigenes used to come up with his system. The author of the 1911 article seemed to think Sosigenes should have noticed this difference, and perhaps he did. I tend to think that if I had to put out user instructions that included division using Roman numerals, I'd have also kept it simple.

But there was indeed an error, and eventually it become noticeable that the spring equinox was no longer occurring at the correct calendar date. By 325, during the conference in Nice, the equinox was on the 21st March and by 1582, when the corrections were finally applied, it had slipped back to the 11th March. Several temporal experts had noticed the discrepancy and had submitted bug reports over the years. But the guys in charge of the calendar, working out of their important seats of power in the Vatican, ignored the reports submitted by Bede, Roger Bacon and others.

Finally, by 1474 things had become so noticeable that Pope Sixtus IV invited the world's foremost astronomer, Regiomontanus, to come to Rome and form an action-oriented team of super techs to solve the problem. Unfortunately, Reggie kicked the bucket before his team was more than half started and the effort degenerated into a memo writing CYA exercise that continued until Pope Gregory XIII saw the light a bit more than a hundred years later.

Gregory twisted arms in the various capitals of Europe and brought in a high powered, highly respected and elderly astronomer named Aloysius Lillius to put together the proclamation which defined the Gregorian calendar. Aloysius (aka Luigi Lilio Ghiraldi) turned to a talented systems analyst named Clavius, who documented the math behind the new system in an 800 page manual called *Romani Calendarii a Gregorio XIII. P.M. restituti Explicatio* (Rome 1603).

When the new calendar was implemented, since one of the goals was to move the spring equinox from the 11th back up to the 21st, Gregory decreed that the day after the feast of St Francis (the 5th of October 1582) would become the 15th of October. And with this implementation, the new method for calculating leap years becomes what we find in Delphi's source code (Listing 1).

So, you might wonder, why did Clavius need 800 pages to say that? As it turns out, Aloysius had a related tasking from Gregory that involved the lunar cycles and the date of Easter. The reason these had to be recalculated was that the system used to work out the lunar cycles under the Julian system didn't work under the Gregorian system. In order to determine the date of Easter, and thereby the dates of the moveable feasts, the following four rules had been established at the council of Nice. First, Easter must be on a Sunday. Then, this Sunday must follow the 14th day of the paschal moon. Thirdly, the paschal moon is that of which the 14th day falls on or next follows the day of the vernal equinox. Lastly, the vernal equinox is fixed invariably in the calendar as the 21st of March.

The system Aloysius came up with to implement this logic yields a method of fixing Easter 'without the possibility of mistake', according to the *Britannica* contributor. Go look it up if you want to learn what an epact is and otherwise thoroughly confuse yourself in three densely packed pages of tables and formulae. Nonetheless, the system is still in effect and is the basis for all the moveable Christian celebrations. The author also points out that this system has the advantage of ignoring the astronomical vernal equinox date, though it apparently does a good job of relating the lunar cycles to the solar year.

However, the fact remains that the function used in Delphi and in Microsoft's software date calculations is not valid for dates prior to 15 October 1582. Suppose your mission as a time patrol person was to prevent interference with the sealing of the Magna Carta on Friday 19 June 1215. If your time machine's temporal navigational system depended on `TDateTimePicker` or `DecodeDate(1215,6,19)`, you would end up arriving ten days too late.

➤ *Listing 2*

```
function tstEncodeDate(Year, Month, Day: Word; var Date: TDateTime): Boolean;
var
  I: Integer;
  DayTable: PDayTable;
const
  MonthDays1752: TDayTable = (31, 29, 31, 30, 31, 30, 31, 31, 20, 31, 30, 31);
begin
  Result := False;
  If year = 1752 then
    DayTable := @MonthDays1752
  else
    DayTable := @MonthDays[IsLeapYear(Year)];
  if (Year >= 1) and (Year <= 9999) and (Month >= 1) and (Month <= 12) and
    (Day >= 1) and (Day <= DayTable^[Month]) then begin
    for I := 1 to Month - 1 do
      Inc(Day, DayTable^[I]);
    I := Year - 1;
    if year <= 1752 then
      Date := I * 365 + I div 4  + Day - DateDelta
    else
      Date := I * 365 + I div 4 - I div 100 + I div 400 + Day - DateDelta;
    Result := True;
  end;
end;
```

Before taking a look at how we might modify `DecodeDate`, we need to take a look at some other gotchas in store for the unwary time travel support engineer.

The foremost gotcha for the English speaking peoples is the fact that our calendar did not change back in 1582. We continued to use the Julian system up to 1750 when, after some opposition, the Calendar Act was enacted so that business and diplomacy with the rest of Europe could be conducted using the same calendar. The Act decreed that the Gregorian system would be used for all public and legal dates, and in order to convert from the Julian system, 'old style', to the 'new style' Gregorian system, Wednesday 2nd September 1752 was followed by Thursday 14th September 1752. In Russia, the Julian system was in use until February 1918, and is apparently still used by the Eastern Orthodox Church. As we go around the world today, we will find everywhere the Gregorian system being used, at least for business. We will also find that when and how the Gregorian system was adopted is unique to almost every country. Sweden, for example, victim of a bureaucratic snafu, has a 30th of February 1712.

If you think this shifting around of the calendar makes the time travel support engineer's job hard, consider the following: in America, George Washington's birthday is celebrated on 22nd February. Perhaps one in ten thousand Americans also happen to know that his actual birthday was 11th February according to the calendar his parents were using during the year he was born, 1732. I doubt if many modern history books bother to let people know that he was really born on the 11th. Thus, our hardy time traveller who blithely uses `EncodeDate(1732, 2, 22)` will show up on the right date, but will go into a panic when he sees a local newspaper that tells him he has arrived on the 11th. Most of the pre-1752 dates a time traveller might aim at, however, will not have been corrected by acts of Congress, and the result will be showing up several days late.

I think a temporal software engineer would want `EncodeDate` to produce a `TDateTime` value that would specify the exact number of days from a reference point to the date in question. `EncodeDate` should shift from Gregorian to Julian methods according to the kind of calendar in operation at the time. `EncodeDate(1752,9,10)`, for example, should raise an exception if we happen to be travelling in the history of the English speaking peoples, but not with French history.

As a time traveller planning my vacation, when I ask my navigational system to show me June 1066, I will be wanting to aim my machine at a Sunday, so that I can arrive when most folks are in church and not likely to notice my arrival. I need the calendar I see to reflect the month as it was then, not as my current, Gregorian, system would think June 1066 was .

Web Sources For Calendar Information

I found dozens of websites with information about calendar systems. Most of them are linked to each other, so you can start exploring just about anywhere. Some seem to be more authoritative than others, but even amongst the most official you will find disagreement. Probably the best FAQ on calendar issues is the *Calendar FAQ* at www.pip.dknet.dk/~c-t/calendar.html. For the Pascal code which I based my own conversion routines on, visit Dr. John Stockton's *Date and Time Miscellany* page at www.merlyn.demon.co.uk/misctime.htm. This page and the related pages and links from his site will give you a good overview of what's happening in the date conscious world as we head to the year 2000, as well as links to most of the places mentioned below. For a detailed set of links, a good place to start is *Calendar Bookmarks* at http://sal.cs.uiuc.edu/~nachum/calendars.html. Another detailed set of links that also includes astronomical stuff is the *Calendrical and Astronomical Links* at www.magnet.ch/serendipity/hermetic/cal_stud/cal_lynx.htm.

For more information on leap seconds and how the atomic clock in Boulder, Colorado, is adjusted, there are some *Proposed answers to selected questions* at www.bldrdoc.gov/timefreq/faq/faq.htm, and a more detailed description from the *International Earth Rotation Service* at http://tycho.usno.navy.mil/time.html. One of the more interesting places I found shows today's date in what seems like over a hundred different calendar systems. It's at www.panix.com/~wlinden/calendar.shtml. Several pages offer conversion facilities. One such is *Calendar Conversions* at http://genealogy.org/~scottlee/calconvert.cgi. Scott Lee's site also has C code for converting to and from a number of different calendar systems. For Java, Lisp and a number of other languages, Nachum Dershowitz and Edward M. Reingold probably have the best site at http://emr.cs.uiuc.edu/home/reingold/calendar-book/index.shtml, where they promote their Cambridge University Press book *Calendrical Calculations*. I'd have liked to read the book, but it's not available in my neck of the woods. I did glance at the Java code, a translation done by Robert C McNally, and found it inspirational. I must confess, however, that even though I've loaded JBuilder and have managed to make a couple of applets work, I still prefer Pascal. For a detailed exploration of calendar history that's very well organized, visit Bill Hollon's site at http://www.greenheart.com/billh/linked.html. He includes a glossary that defines most terms associated with calendars you are likely to come across.

A good reference for Julian Day Numbers is Peter Meyer's page at http://www.magnet.ch/serendipity/hermetic/cal_stud/jdn.htm, which is associated with the calendar and astronomical link page I mentioned earlier. What I find fascinating about this page is the bottom, where the algorithm for converting between Gregorian and Julian Day Numbers is spelled out. With different constants than those defined in *Numerical Recipes in Pascal*. A visit to the SWAG site at http://www.gdsoft.com/swag/swag.html will produce a number of Pascal procedures and functions, each of which seems to use a different set of constants.

```
Type
  TCalendarDate = record
    Year, Month, Day : integer;
  end;
  TMJD = double;  // This will be our baseline number.
  TLinkDate = record
    //a date in the current calendar and its MJD equivalent
    Date : tCalendarDate;
    MJD : TMJD;
  end;
  TGregorianChangeRec = record
    // MJD value of last date in in  system, first in new
    LastMJD : TMJD;
    // Last date in old system, first date of new system
    LastDate : tCalendarDate;
    // days +/- to add/delete to/from calendar
    Adjustment : integer;
    // day of month/year + adjustment =
    // first date Gregorian system in use
  end;
  TmonthStructure = class
    // convenient way to ensure we have a place
    // to put unique months
    private
      fNumDays : integer;
      fName : string;
      fMissingDaysStart,
      fMissingDaysEnd : integer;
    public
      constructor BuildMonth(const aName : string;
        const aNumDays, StartMissing, EndMissing : integer);
      function HasMissingDays(var First, Last : integer):
        boolean;
      Property NumDays : integer
        read fNumDays write fNumdays;
      property Name : string read fName write fName;
      Property FirstMissingDay : integer
        read fMissingDaysStart write fMissingDaysStart;
      Property LastMissingDay : integer
        read fMissingDaysEnd write fMissingDaysEnd;
  end;
  TYearStructure = class
    // a class that ensures we can specify unique year
    // structures.
    private
      fNumMonths : integer;
      fMonthList : tlist;
      procedure GrowMonthList;
    protected
      function GetMonthName(index : integer): string;
      procedure SetMonthName(index : integer;
        aName : string);
      function GetMonthLen(index : integer): integer;
      procedure SetMonthLen(index : integer;
        aLen : integer);
      function GetMonthStruc(index : integer):
        tMonthStructure;
      procedure setMonthStruc(index : integer;
        aStruc : tMonthStructure);
    public
      constructor create;
      destructor destroy; override;
      property NumMonths : integer
        read fNumMonths write fNumMonths;
      property MonthName[index : integer] : string
        read getMonthName write setMonthName;
      Property MonthLen[index : integer] : integer
        read getMonthLen write setMonthLen;
      Property MonthObj[index : integer] : tMonthStructure
        read getMonthStruc write setMonthStruc;
  end;
```

➤ *Listing 3*

```
TCalendarDef = class
  Private
    fName : string;     // name, e.g. English, Swedish, Roman
    fDate : TLinkDate;  // Date we are currently working with
    fAstro : boolean;   // set true to insert a year zero between 1BC and 1AD
    fYearDef : TYearStructure;
    fDayName : TStringlist;  // count is the number of days per week
    fDayStart : double;      // 0.0 = midnight, 0.5 = noon, etc.
    fIsLeapYear : tLeapYearRule;
    fOnEncode : tEncodeDateProc;
    fOnDecode : tDecodeDateProc;
    fAlignmentDate : TLinkDate;
    // name of calendar system before fGregorian date
    fNameOfPreviousSystem : string;
    fGregorianDate : TGregorianChangeRec;
    // true means use previous system for dates before Gegorian dates
    // ie do not use proleptic calculations
    fSwitchOnChangeDate : boolean;
  Protected
    function getDate : TLinkDate; virtual; abstract;
    procedure setDate(avalue : tLinkDate); virtual; abstract;
  ...
```

➤ *Listing 4*

SEPTEMBER 1752						
S	M	Tu	W	Th	F	S
		1	2	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

➤ *Figure 2*

To get this kind of calendar, we'll need to add branching logic into the existing routines. For example, we could modify `EncodeDate` as shown in Listing 2.

When testing this, however, I did not come up with the results I expected. 11 February 1732 gives me a value of -61305 rather than the -61307 I think I should get. As I started looking for what might be causing the discrepancy, it occurred to me that this approach is not very flexible anyway. My time machine lets me move in space as well as time, so how do I deal with destinations before 1582 in Italy or before 1712 in Sweden?

At first glance, the approach used by Dr. Stockton in MJD_DATES.PAS provides the basic flexibility we need. He has written a group of functions for converting Julian to Gregorian dates by passing them through an MJD baseline, established by the fact that MJD 50,000 is 10 October 1995. Scott Lee has written a group of C programs that use a similar approach using what he calls Standard Day Numbers, which are essentially the same as regular Julian Days. Lee's conversion routines also cover Jewish, Islamic and other calendar systems. See the box on web sources for where to find these programs.

In the spirit of Delphi, however, I'm looking for something a bit more than conversion functions, despite the fact that `EncodeDate` and `DecodeDate` are themselves simply functions in SYSUTILS.PAS. What I'm looking for is something more along the lines of `TDateTimePicker` that will correctly handle September 1752. So that it will show up looking something like Figure 2.

Now we get to the place where I have to ask my unknown users, both component users and their end users, how should we handle dates before the changeover? Time travellers, historians and genealogists would probably want to see the Julian calendar date. Should we have some kind of graphic or symbol to alert users that we changed the way we are computing dates? If you are in an Islamic country, or Israel, or China, you probably want recent dates in the same Gregorian system the rest of us are using, but when you are scrolling a calendar back into the past, would you want the calendar structure, month names, etc.

to change when you got past a certain point?

To achieve the kind of flexibility we need to be able to handle any set of answers to these questions, object orientation can help us. There is a set of Java programs that may perhaps contain the kind of structure we are looking for. Based on the work of Nachum Dershowitz and Edward M. Reingold in their book *Calendrical Calculations*, published by Cambridge University Press, the Java class hierarchy was created by Robert C McNally. See the web sources boxout.

I'm not sure if I want to tackle translating what McNally has done to Delphi or not.

I'm going to propose a basic set of classes for flexible calendars and discuss some of the code needed to implement them.

The first group of types merely set up some convenient structures for later use, see Listing 3.

To provide a home base for our calendar systems, the following class is designed to be inherited from rather than instantiated. `TCalendarDef` abstracts the facts surrounding a change from one method of computing dates to another by completely defining the year and month being worked with. For any particular culture, then, the facts surrounding that culture's shift from one calendar to another can be expressed by typed constants or by modifying the descendant class at runtime, see Listing 4 (I've left out the redundant private get and set methods to save space here). Perhaps Inprise could consider setting up Delphi so that read and write portions of property declarations are in fact forward method declarations. Sure would cut down on the typing, especially if there was a button to generate the stubs (Listing 5).

To define the English calendar, all we need are a few typed constants and a descendant of `TCalendarDef`. However, keep in mind that, for example, if you were to derive a component for the Jewish calendar, you could not use typed constants since the change date aspect of this would need to be a runtime phenomenon: in Israel, the official calendar is not Gregorian, but almost all businesses with any international connection do use the Gregorian calendar. So an Israeli calendar component would have to allow for shifting the change date to any date: see Listing 6.

To instantiate the English calendar, all we need is the constructor implementation shown in Listing 7.

The real work of the new calendar system is in the `Encode` and `Decode` methods, which are to be found on this month's disk in the file BASEDATE.PAS, where I've implemented the above described classes.

As my test project shows (see Figure 1), the component works: it gives us an error for the missing dates in September 1752, and automatically shifts to Julian for dates prior to that as shown when you move the numbers over to the standard Delphi `TDateTime` and back again. The code on the disk also includes the complete class

```
  Procedure fEncodeDate(var MJD : TMJD; const aYear, aMonth, aDay :integer);
    virtual;  abstract;
  Procedure fDecodeDate(const MJD : TMJD; var aYear, aMonth, aDay :integer);
    virtual;  abstract;
  Property CalendarName : string read fName write fName;
  Property OldCalendarSystemName : string
    read fNAmeOfPreviousSystem write fNameOfPreviousSystem;
  Property ShowPreviousDatesInPreviousSystem : boolean
    read fSwitchOnChangeDate write fSwitchOnChangeDate;
  Property Astro : boolean read fAstro write fAstro default false;
  Property DaysPerYear : cardinal read GetDaysPerYear; // write SetDaysPerYear;
  Property NumberOfMonths : integer
    read getNumberOfMonths write setNumberOfMonths;
  Property MonthName[index : integer] : string
    read getMonthName write setMonthName;
  Property MonthLength[index : integer] : cardinal
    read getMonthLength write setMonthLength;
  Property DayName[index : integer]: string
    read getDayName write setDayName;
  Property DayStart: double
    read getDayStart write setDayStart;
  Property AlignmentDate : tLinkDate
    read getAlignmentDate write setAlignmentDate;
  Property ChangeDate : TGregorianChangeRec
    read getChangeDate write setChangeDate;
Public
  constructor create; virtual;
  destructor destroy; override;
  Function IsLeapYear(aYear : integer): boolean; virtual; abstract;
  Function EncodeDate(const aYear, aMonth, aDay :integer): tMJD;
    virtual;  abstract;
  Function DecodeDate(const MJD : TMJD): tCalendarDate; virtual; abstract;
  Function MSDatefromMJD(const MJD : TMJD): tDateTime; virtual; abstract;
  Function MJDfromMSDate(const aDateTime : tDateTime): TMJD; virtual; abstract;
end;
```

➤ *Listing 5*

definitions and a modified calendar display component, along with a test project for illustrating what we've learned about the calendar in this article.

However, there are many factors I did not build into my code. Most of them are listed in the boxout entitled *Problems Not Solved*.

A few comments about the code are in order. The algorithms I used for Encode and Decode are directly derived from Dr Stockton's work (see web sources boxout and the HTML link page I've included on this month's disk). However, I was able to directly use very little of his actual code for three reasons. The most important reason is that I am using properties at several points rather than the typed constants he used. The second reason is that Delphi 3 seems to be a lot stricter about parameter passing than Turbo Pascal 7 was. The most trivial reason, and the most frustrating, was that when I pasted from the text file I downloaded into the Delphi IDE, something apparently went wrong with the line endings. I kept getting spurious errors until I exited Delphi, pulled up the Pascal source file in my text editor and used the end of line override function to save it back with a CRLF at the end of each line.

I got the YMD to MJD method to work in fairly short order, though I was puzzled at first by why his month array started with March instead of January. Going the other way had me stumped for several days, however. I backed out of his logic and implemented the logic I found in *Numerical Recipes for Pascal*, which worked, but was off by one day because it is based on regular Julian Day Numbers, which start at noon. The *Numerical Recipes* logic is included in the source code, commented out. I took a break and, upon my next attempt, got Stockton's logic to work.

What always amazes me is how two such different approaches come up with the same answer. The secret is in the constants, I suppose. Delphi's constants and

```
const
  cEnglishLinkDate : tLinkDate =
    (Date: (year: 1995; month : 10; day: 10); MJD: 50000);
  cEnglishChangeDate : tGregorianChangeRec =
    (LastMJD : cLastMJDEnglish; LastDate: (year: 1752; month: 9; day: 13);
    Adjustment: 11);
  cNormalMonthLengths : array[1..12] of integer =
    (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
Type
  TEnglishCalendar = class(tCalendarDef)
    Private
      fSeptember1752,
      fNormalSeptember : tMonthStructure;
    Protected
      function getDate : TLinkDate; override;
      procedure setDate(aValue : tLinkDate); override;
      Property LeapYearRule : tLeapYearRule read fIsLeapYear write fIsLeapYear;
    Public
      Constructor create; override;
      Destructor destroy; override;
      Function IsLeapYear(aYear : integer): boolean; override;
      Function EncodeDate(const aYear, aMonth, aDay :integer): tMJD; override;
      Function DecodeDate(const MJD : TMJD): tCalendarDate; override;
      Function MSDatefromMJD(const MJD : TMJD): tDateTime; override;
      Function MJDfromMSDate(const aDateTime : tDateTime): TMJD; override;
      Property CalendarName;
      Property OldCalendarSystemName;
      Property ShowPreviousDatesInPreviousSystem;
      Property Astro;
      Property DaysPerYear;
      Property NumberOfMonths;
      Property MonthName;
      Property MonthLength;
      Property DayName;
      Property DayStart;
      Property AlignmentDate;
      Property ChangeDate;
    end;
```

➤ *Above: Listing 6*

➤ *Below: Listing 7*

```
Constructor TEnglishCalendar.create;
var
  i : integer;
begin
  inherited create;
  SetAlignmentDate(cEnglishLinkDate);
  SetChangeDate(cEnglishChangeDate);
  FYearDef.NumMonths := 12;
  for i := 1 to 12 do
    FYearDef.MonthObj[i] := tMonthStructure.buildMonth(longMonthNames[i],
      cNormalMonthLengths[i], 0, 0);
  fSeptember1752 := tMonthStructure.buildMonth(LongmonthNames[9], 19, 3, 13);
  fNormalSeptember := tMonthStructure.buildMonth(LongmonthNames[9], 30,0,0);
  for i := 1 to 7 do
    FDayName.add(LongDayNames[i]);
  fSwitchOnChangeDate := true;
  fNameOfPreviousSystem := 'Julian';
  fName := 'English';
end;
```

Stockton's constants are very similar in that they are both sets of integers and specify the number of days in different types of years. The *Numerical Recipes* constants are of type real and are simply inserted inline, they reflect an entirely different approach. You will find several other methods for doing these calculations, most of them using a different set of constants.

Constants in algorithms and formulae have always bothered me. I'm sure there is a good reason for a line like the following one extracted from the middle of the *Numerical Recipes* code:

```
je := trunc((jb-jd) / 30.6001);
```

It's been my experiance that the reason is usually something like 'because it works.' When I go on to ask why, I usually get that stare reserved for ditch diggers and infants. So now, in my mature years, I just use what I find and if it doesn't work, I look elsewhere. That still doesn't stop me from

Year 2000 Concerns

I had been blithely going along ignoring the whole year 2000 hullaballo, knowing that my Delphi creations were using TDateTime and therefore were faithfully storing the century along with the last two digits of the year. Not that I haven't been hoping for some work to come my way converting someone's old dBase file, or perhaps someone's Turbo Pascal 3 program which ignored the century by being clever with strings. But the real problem, I'd always assumed, was monster applications written in COBOL back in the 50s and 60s. I can wade through COBOL code, but no way can I call myself a COBOL programmer.

However, what I didn't realize was that there are built-in bugs in many of the hardware platforms currently being used. For example, John Stockton (see the web sources boxout) points out that 'In many cases, the clock chip and ROM BIOS of a PC will not correctly handle the 1999 to 2000 transition, going to a date in 1980; but will work properly thereafter, after the date has been corrected once ...'

If you are into defensive coding, you'll need to look into this area. Some platforms will have machine code controlling them which has been assembled to think the year 2000 will not be a leap year. Who knows, the Bios controlling your PC may have been coded by someone who watched the Opra Winfrey show when her audience decided the year 2000 would have a February 30th.

wondering where on earth these constants come from.

Conclusions

The basic fact you should have learned from this article is that the calendar is not a fixed entity and that Delphi's date and time functions and procedures only work within a limited time span. By implication, you should also have learned that should you put a date field in an application targeted at cultures other than your own, you may have much more than Windows' international settings to contend with, especially if a user might need to use your application to work with historical events. In the meantime, I'd welcome any suggestions for generalizing `TCalendarDef` to handle other systems, as well as a chance to see some specific descendants at work.

Brandon Smith is currently working for Rose International, doing Delphi and web development. He can be reached by email at delphi@synature.com or visited at www.synature.com

Problems Not Solved

Here are the issues not solved in my component:

1. The actual year, the number of days between one equinox and the next, varies. According to the Calendar FAQ (see web sources boxout), around 1900 the year was 365.242196 days. By about 2100 it will be 365.242184 days. The Julian system assumes every year is 365.25 days. The Gregorian system assumes every year is 365.2425 days. There is apparently a regular cycle of number of days in the year that repeats every 21,000 years.

2. However, the leap second adjustments made to the atomic clocks are based on observation, not mathematical calculations, of the earth's orbit. As the Naval Observatory in Washington DC puts it, 'The earth is constantly undergoing a deceleration caused by the braking action of the tides. [...] Other factors also affect the earth, some in unpredictable ways, so that it is necessary to monitor the earth's rotation continuously.' So the world's atomic clocks have been adjusted by a second approximately every 500 days since 1972. So far, each of the adjustments have been to add a second. One of the main reasons the world's chronological authorities, such as the International Earth Rotation Service, adjust clocks to the actual rotation of the earth is so that the satellites used for the Global Positioning Systems are accurate with regard to the signals they send down to the bass fisherman trying to get to that spot in the lake where he was last time.

3. These two factors make any use of constants in our date conversion algorithms a bit suspect, at least for a time traveller who wishes to arrive on the date selected. If I were going back to 50,000 BC, I don't think I'd care much if I arrived on a Tuesday when I was shooting for Friday. But I would be upset if I was shooting for summer and ended up in winter.

4. The day added to make a leap year is not really 29 February, but rather the 24th. I'm not sure this is worth pursuing, though if a time traveller is trying to go back to a date expressed as a name day rather than a calendar date, and that name day is associated with a date between 23 and 29 February, this fact becomes critical.

5. From 46 BC to 12 AD the arrangement of months with the year and the number of days per month shifted around quite a bit. It would appear the information is available to classical scholars and that special `TMonthStructure` and `TYearStructures` could probably be added as typed constants to describe this period. However, the `IsLeapYear` function would also need special logic to handle the period when the leap years were added every third year, then skipped entirely to make up the extra gain.

6. Nothing in this component is set up for handling calendar systems based on the lunar cycles, such as the Jewish system. A `TLunarStructure` could surely be devised to handle the task. However, the actual length of a lunar cycle also varies over time. In addition, there are some modern Islamic countries where the start of the month is determined by actual observation of the moon rather than by a precalculated system. I'm not sure I'd care to attempt writing the code to support a forward travelling time machine in one of those countries.

7. New Year's Day has also varied over the centuries. Thus, it is quite possible to find a document dated 2 February 1306 which refers to 2 June 1307 and means four months from then, not a year and four months, since the year changeover was to occur on 1 March, or 25 March, or the Saturday before Easter. The `TYearStructure` is not currently set up to handle this kind of variation in how a year is defined. A time traveller trying to be at the signing of a document would not only have to know which culture he was going to, but also what kind of document. For example, the Calendar FAQ points out that:

'In England (but not Scotland) three different years were used. The historical year, which started on 1 January. The liturgical year, which started on the first Sunday in Advent. The civil year, which from the 7th to the 12th century started on 25 December, from the 12th century until 1751 started on 25 March, from 1752 started on 1 January.'

8. `TDateTimePicker` doesn't always work if the `Date` property is set to anything less than zero. It doesn't error off, and the date value is accepted, but sometimes it doesn't show the new date.

9. `TDateTime` will not work at all for dates prior to 1/1/1 in the Proleptic Gregorian Calendar, nor will it work for dates after 12/31/9999. This means that the various formatting and conversion routines won't work beyond those limits either. The `TCalendarDef` class does not include replacements for formatting dates, though it will convert back and forth between MJD and `TDateTime` within the limits of `TDateTime`. However, conversions of dates before the calendar reformation may not give you what you are expecting.