NEWS · AUGUST 14, 2026

Four German Holidays Hang Off One Number

Part two of a two-part series on computing public holidays in Delphi — the Gauss/Meeus Easter algorithm in fifteen lines, the four German holidays derived from it by simple offsets, and merging two national calendars into one with the rules-as-data structure from Part 1.

By Dr. Holger Flick



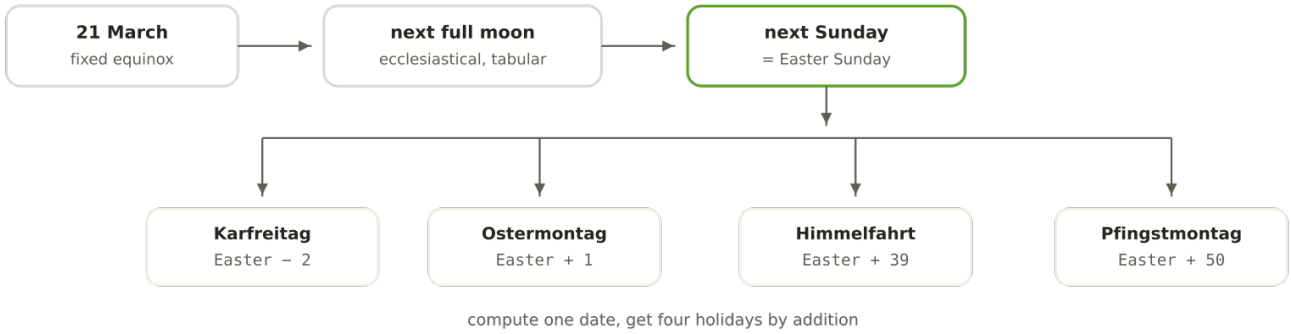
Part 1 built a holiday engine out of data instead of branches: a record holding a stable identifier, a display name, and an anonymous method that turns a year into a date. Eleven US federal holidays fit into that structure with three small primitives — a fixed date, the *n*th weekday of a month, and the last weekday of a month — and the code that evaluates them never branches on rule type at all.

Germany is where that design gets tested, because the German calendar contains a kind of rule the US calendar doesn't have. Four of the nine nationwide German holidays aren't anchored to the Gregorian calendar at all. They're anchored to Easter — and Easter is not a date you can look up in a statute. It's the result of a calculation that reconciles a solar calendar with a lunar one, standardized at the Council of Nicaea in 325 AD, and it moves across a 35-day window from year to year.

The good news is that the whole thing fits in about fifteen lines. This post implements Easter, adds the German rule table, handles the fact that the two countries treat weekend holidays completely differently, and merges both calendars into one — all without touching the evaluation loop from Part 1. That last part is the real point: a good data structure absorbs a genuinely new kind of rule without a rewrite.

Why Easter moves

Easter is the first Sunday after the first ecclesiastical full moon on or after 21 March. Every clause in that sentence is doing work: the *ecclesiastical* full moon is a tabular approximation rather than an astronomical observation, 21 March is a fixed stand-in for the equinox, and the "first Sunday after" is what pins the result to a weekday. The consequence is that Easter can fall anywhere from 22 March to 25 April.



Easter's definition chains three conditions — and four German holidays chain off the result

The diagram is the reason this is worth the effort: Easter is the only genuinely hard computation in the German calendar, and once you have it, four holidays are one `IncDay` call each. The offsets are fixed — they never vary by year — so the hard part is solved exactly once.

The algorithm, in fifteen lines

The standard way to compute Gregorian Easter is the *anonymous Gregorian algorithm*, usually credited to Gauss and refined into the form below by Butcher and Meeus. It's a sequence of integer divisions and remainders that reconstructs the lunar cycle, and it's valid for any year in the Gregorian calendar. I'm reproducing it as documented on Wikipedia, keeping the traditional single-letter variable names because they make it checkable against every other published implementation:

```
/// Easter Sunday (Western/Gregorian) for AYear.
/// Anonymous Gregorian algorithm (Gauss / Meeus-Jones-Butcher).
function EasterSunday(const AYear: Word): TDate;
var
  a, b, c, d, e, f, g, h, i, k, l, m, Month, Day: Integer;
begin
  a := AYear mod 19;
  b := AYear div 100;
  c := AYear mod 100;
  d := b div 4;
  e := b mod 4;
  f := (b + 8) div 25;
  g := (b - f + 1) div 3;
  h := (19 * a + b - d - g + 15) mod 30;
  i := c div 4;
  k := c mod 4;
  l := (32 + 2 * e + 2 * i - h - k) mod 7;
  m := (a + 11 * h + 22 * l) div 451;
  Month := (h + l - 7 * m + 114) div 31;         // 3 = March, 4 = April
  Day   := ((h + l - 7 * m + 114) mod 31) + 1;
  Result := EncodeDate(AYear, Month, Day);
end;
```

I won't pretend the individual steps are intuitive — they're a compressed reconstruction of the Metonic cycle, and the intermediate values don't correspond to anything you'd recognize as a date. What matters practically is that it's deterministic, dependency-free, and testable against known values. Easter 2026 is 5 April, 2027 is 28 March, and 2024 was 31 March; if your implementation reproduces those three, you've almost certainly typed it in correctly.

> **This computes *Western* Easter only**
> Most Orthodox churches calculate Easter from the Julian calendar, which usually yields a different date — in 2026 Western Easter is 5 April while the Orthodox date is 12 April, and in 2027 they diverge by five weeks. The algorithm above is the Gregorian one, which is what German public holidays are based on. If you need the Orthodox date, that's a different computation, not a parameter to this one.

The four derived holidays are then a matter of adding days, which is what `IncDay` from `System.DateUtils` is for:

```
function EasterOffset(const AYear: Word; const ADays: Integer): TDate;
begin
  Result := IncDay(EasterSunday(AYear), ADays);
end;
```

The offsets are worth stating precisely because two of them are traditionally quoted as different numbers. Christi Himmelfahrt is *Easter + 39*, not + 40, even though it's called the fortieth day after Easter — the ancient convention counts Easter Sunday itself as day one. Pfingstmontag is Easter + 50 for the same reason: Pentecost is the fiftieth day counting inclusively, landing it on Easter + 49, and the Monday after is + 50.

The German table: nine holidays every state agrees on

Germany's holiday law works differently from the American federal model, and the difference is worth getting right rather than glossing over. Holidays are set by the *states* — each Bundesland has its own Feiertagsgesetz — so there is no single federal list. What exists instead is an overlap: nine days that all sixteen states independently designate. Those are the *bundeseinheitliche Feiertage*, and they're the honest scope for a general-purpose calendar.

The one genuine exception is the Tag der Deutschen Einheit, which is fixed by federal law in Article 2(2) of the Einigungsvertrag, the 1990 reunification treaty. Everything else on the list is state law that happens to agree everywhere:

```
function GermanRules: TArray<THolidayRule>;
begin
  Result := [
    THolidayRule.Create('neujahr', 'Neujahrstag',
      function(const Y: Word): TDate begin Result := Fixed(Y, 1, 1); end),
    THolidayRule.Create('karfreitag', 'Karfreitag',
      function(const Y: Word): TDate begin Result := EasterOffset(Y, -2); end),
    THolidayRule.Create('ostermontag', 'Ostermontag',
      function(const Y: Word): TDate begin Result := EasterOffset(Y, 1); end),
    THolidayRule.Create('tag-der-arbeit', 'Tag der Arbeit',
      function(const Y: Word): TDate begin Result := Fixed(Y, 5, 1); end),
    THolidayRule.Create('christi-himmelfahrt', 'Christi Himmelfahrt',
      function(const Y: Word): TDate begin Result := EasterOffset(Y, 39); end),
    THolidayRule.Create('pfingstmontag', 'Pfingstmontag',
      function(const Y: Word): TDate begin Result := EasterOffset(Y, 50); end),
    THolidayRule.Create('tag-der-deutschen-einheit', 'Tag der Deutschen Einheit',
      function(const Y: Word): TDate begin Result := Fixed(Y, 10, 3); end),
    THolidayRule.Create('erster-weihnachtstag', '1. Weihnachtstag',
      function(const Y: Word): TDate begin Result := Fixed(Y, 12, 25); end),
    THolidayRule.Create('zweiter-weihnachtstag', '2. Weihnachtstag',
      function(const Y: Word): TDate begin Result := Fixed(Y, 12, 26); end)
  ];
end;
```

Nine rules, four of which are Easter offsets, and the record from Part 1 accepted the new rule type without a single change. That's the design working: `EasterOffset` is just another `function(Year): TDate`, structurally indistinguishable from `Fixed` as far as the table is concerned.

> **The state-level holidays this list leaves out**
>
> Several widely observed German holidays are deliberately absent because they're not nationwide: Heilige Drei Könige (6 January) is a holiday in three states, Fronleichnam in Bavaria and a handful of others, Allerheiligen in the predominantly Catholic states, and Reformationstag across most of the north and east. Frauentag is a holiday in Berlin and Mecklenburg-Vorpommern. Every one of them is a real day off for millions of people — they're just not universal, and a calendar claiming to be *the* German list while including them would be wrong for most of the country. If you need state-level accuracy, the natural extension is a `States: TArray<string>` field on `THolidayRule` and a filter in `ComputeYear`; the data structure takes it without complaint.

Running the table for 2026 gives the nine dates below, with the Easter-derived four clustered in spring exactly as the offsets predict:

```
Neujahrstag                Thu, 01 Jan 2026
Karfreitag                 Fri, 03 Apr 2026
Ostermontag                Mon, 06 Apr 2026
Tag der Arbeit             Fri, 01 May 2026
Christi Himmelfahrt        Thu, 14 May 2026
Pfingstmontag              Mon, 25 May 2026
Tag der Deutschen Einheit  Sat, 03 Oct 2026
1. Weihnachtstag           Fri, 25 Dec 2026
2. Weihnachtstag           Sat, 26 Dec 2026
```

Karfreitag on 3 April and Ostermontag on 6 April bracket Easter Sunday on the 5th, and Christi Himmelfahrt is a Thursday — as it always is, being Easter Sunday plus 39 days. Those are useful invariants to assert in a test.

Where the two countries genuinely differ: weekends

Look at the German list again and note Tag der Deutschen Einheit falling on a Saturday in 2026, and the 2. Weihnachtstag doing the same. In Germany, that's simply a lost holiday — German law provides no mechanism for moving a holiday that lands on a weekend, and workers do not get a substitute day. It's a recurring topic of public grumbling in years like this one.

The United States does the opposite. Under 5 U.S.C. § 6103(b), a holiday falling on a Saturday is observed on the preceding Friday, and one falling on a Sunday is observed on the following Monday. So Independence Day 2026, a Saturday, is observed by federal employees on Friday 3 July.

This is the one place where a country-level flag is genuinely warranted, and the clean way to model it is to keep *occurrence* and *observation* as separate fields rather than overwriting one with the other:

```
type
  TWeekendRule = (wrNoShift, wrShiftToNearestWeekday);

  THolidayCalendar = record
    CountryCode: string;
    Rules: TArray<THolidayRule>;
    Weekend: TWeekendRule;
  end;

function ObservedDate(const ADate: TDate; const ARule: TWeekendRule): TDate;
begin
  Result := ADate;
  if ARule = wrNoShift then
    Exit;
  case DayOfTheWeek(ADate) of
    DaySaturday: Result := IncDay(ADate, -1);   // observed the Friday before
    DaySunday:   Result := IncDay(ADate, 1);    // observed the Monday after
  end;
end;
```

Keeping both dates means you can answer both questions from one table: "is today Christmas?" uses the occurrence date, and "is the office closed?" uses the observed one. Collapse them into a single field and you permanently lose the ability to ask the first question — which matters more than it sounds, because anniversary and age calculations want the real date, not the administrative one.

> **The shift can move a holiday into the previous or next year**
>
> New Year's Day 2028 falls on a Saturday, so US federal practice observes it on Friday 31 December 2027. A calendar generated for 2027 alone won't contain it, and a calendar for 2028 will place it outside the year it belongs to. If you're generating a strict calendar-year range, compute the neighboring years too and filter afterward — computing three years and discarding two is cheap, and getting this wrong produces an off-by-one-day error at exactly the moment the accounting year rolls over.

Merging both calendars

With both tables defined, a combined calendar is a fold over the countries, and the dictionary from Part 1 does the deduplication:

```
type
  TObservedHoliday = record
    UID: string;
    Name: string;
    CountryCode: string;
    Occurs: TDate;      // the actual calendar date
    Observed: TDate;    // the day off, after any weekend shift
  end;

function ComputeCalendars(const ACalendars: TArray<THolidayCalendar>;
  const AYear: Word): TArray<TObservedHoliday>;
var
  Cal: THolidayCalendar;
  Rule: THolidayRule;
  Item: TObservedHoliday;
  List: TList<TObservedHoliday>;
begin
  List := TList<TObservedHoliday>.Create;
  try
    for Cal in ACalendars do
      for Rule in Cal.Rules do
      begin
        Item.UID         := Rule.UID;
        Item.Name        := Rule.Name;
        Item.CountryCode := Cal.CountryCode;
        Item.Occurs      := Rule.Compute(AYear);
        Item.Observed    := ObservedDate(Item.Occurs, Cal.Weekend);
        List.Add(Item);
      end;
    List.Sort(TComparer<TObservedHoliday>.Construct(
      function(const L, R: TObservedHoliday): Integer
      begin
        Result := CompareDate(L.Occurs, R.Occurs);
        if Result = 0 then
          Result := CompareStr(L.CountryCode, R.CountryCode);
      end));
    Result := List.ToArray;
  finally
    List.Free;
  end;
end;
```

`CompareDate` rather than a raw `<` comparison is the Part 1 habit paying off — it compares the date portion only, so a stray time fraction on any `TDate` can't scramble the sort order. The country code as a tiebreak keeps the output stable when two holidays share a date, which brings us to the interesting collision.

Run both calendars for 2026 and two dates carry entries from both countries. 1 January is the obvious one — New Year's Day and Neujahrstag are the same day everywhere. The other is 25 December, Christmas Day and 1. Weihnachtstag. And there's a subtler near-miss worth noticing: **Memorial Day 2026 and Pfingstmontag**

2026 both fall on Monday 25 May — the last Monday in May colliding with a date derived from Easter, a

coincidence that holds in 2026 and breaks in 2027, when Pfingstmontag is 17 May and Memorial Day is the 31st.

That collision is precisely why the index in Part 1 used `AddOrSetValue`. Build a `TDictionary<TDate, string>` over a merged calendar with plain `Add` and 2026 will raise a duplicate-key exception on 25 May — in production, in the one year it happens. A multi-value index is the honest structure once more than one country is in play:

```
function MergedIndex(const ACalendars: TArray<THolidayCalendar>;
  const AYear: Word): TDictionary<TDate, TArray<string>>;
var
  H: TObservedHoliday;
  Existing: TArray<string>;
begin
  Result := TDictionary<TDate, TArray<string>>.Create;
  for H in ComputeCalendars(ACalendars, AYear) do
  begin
    if not Result.TryGetValue(H.Observed, Existing) then
      Existing := [];
    Result.AddOrSetValue(H.Observed, Existing + [H.CountryCode + ': ' + H.Name]);
  end;
end;
```

Keying on `Observed` rather than `Occurs` is the right default here because the question this index answers is "is anyone off work today?" Swap the field if you're asking the calendrical question instead — and the fact that swapping one identifier changes the meaning cleanly is the last dividend of having kept both dates.

Putting it to work: business days

The payoff for all of this is usually a business-day calculation, which is now short enough to write inline. Note that it needs holidays from every year the range touches, which is the practical reason `ComputeYear` takes a year rather than caching a single global:

```
function AddBusinessDays(const AStart: TDate; const ACount: Integer;
  const AHolidays: TDictionary<TDate, TArray<string>>): TDate;
var
  Remaining: Integer;
begin
  Result := DateOf(AStart);
  Remaining := ACount;
  while Remaining > 0 do
  begin
    Result := IncDay(Result, 1);
    if (DayOfTheWeek(Result) in [DayMonday..DayFriday]) and
       (not AHolidays.ContainsKey(Result)) then
      Dec(Remaining);
  end;
end;
```

`DayOfTheWeek` being ISO-numbered is what makes `[DayMonday..DayFriday]` a legible range rather than a puzzle — with the Sunday-first `DayOfWeek` from SysUtils, the working week would wrap awkwardly around the set boundary. Small thing, but it's the difference between a line you read and a line you decode.

> **When to reach for a library instead**
>
> This series builds about a hundred and twenty lines that you fully understand, which is the right trade for one or two countries. It stops being the right trade quickly. If you need state-level German rules, US state holidays, or any third country, the maintained open-source options are the better answer: python-holidays and Nager.Date both cover subdivisions across a hundred-plus countries, and date-holidays does the same in JavaScript. They also track the thing hand-rolled code never does — rule *changes*, like Juneteenth being added in 2021 or a one-off national holiday declared for a state funeral. For Delphi specifically, TZDB is the equivalent for the time-zone half of the problem and is worth having regardless.

Takeaways

Two posts, two countries, and the evaluation loop never changed once.

- **Easter is the only hard computation in either calendar**, and it's fifteen lines of integer arithmetic with no dependencies. Four German holidays are then simple offsets from it.
- **The offsets are +39 and +50**, not +40 and +49 — the traditional counting includes the starting day, and off-by-one here is silent.
- **Germany's nine nationwide holidays are an overlap, not a federal list.** Only the Tag der Deutschen Einheit is set federally; the other eight are sixteen state laws that agree.
- **Occurrence and observation are different dates.** The US shifts weekend holidays to the nearest weekday; Germany never shifts anything. Keep both fields and you can answer both questions.
- **Two holidays can share a date** — Memorial Day and Pfingstmontag collide on 25 May 2026 — so any index over a merged calendar needs to expect it.

> The hard part of a holiday calendar isn't the algorithm, it's the modeling. Get "a holiday is a function from a year to a date" right and Easter, weekend shifts, and a second country all slot in without a rewrite.

If you want the groundwork underneath all of this — what a `TDateTime` actually is, and the `System.DateUtils` functions these two posts lean on throughout — the Dates and Times in Delphi tutorial covers it in two parts, including the time-zone discipline that matters the moment your holiday calendar is consumed by a client in another country.

> **The series: Computing Public Holidays in Delphi**
> Two parts: 1. Rules as data, and the US federal calendar 2. Easter, the German calendar, and merging two countries **— you are here** Both build on the Dates and Times in Delphi tutorial.

INDEPENDENT · AD-FREE · READER-SUPPORTED

Free to read. Not free to make.

Every tutorial, article, and line of example code on Holger's Code is published without a paywall and without ads. This page is about what keeps it that way.

Professional work, published free

The content here is held to the same standard as professional client work: every tutorial is built as a real project, tested against current versions, documented step by step, and revisited when the frameworks move on. A single in-depth tutorial routinely takes days — and that's before the bills behind it: servers, storage, build infrastructure, licenses, and the hardware everything is verified on.

If an article or tutorial here has saved you an afternoon of debugging, taught you a technique, or helped you ship, it created real value for you. Supporting it isn't charity — it's paying for professional work you found useful, at whatever price you choose. And if you can't, keep reading anyway. That's what the free part means.

Ways to support

KO-FI

Buy me a coffee

A one-time tip — no account, no subscription, any amount.

BOOKS & COURSES

Buy the paid work

The paid tier of everything published free here — and you get something lasting in return.

SPREAD THE WORD

Share an article

Send a tutorial to a colleague, cite an article, subscribe to the RSS feed.

Nothing here goes behind a paywall, whether you chip in or not. But if something on this site earned its keep today, this is where to say so.

Support on Ko-fi —