Timely

A time zone module

Version 4,000,000

J. M. van Schalkwyk

Contents

1.0 A time zone module

This Perl module is fairly general in its scope, but designed to be used with two specific programs. The first is an intialisation/testing program written in Perl (*small.pl*); the second is a much larger Perl program called vector seekwell that is used to interrogate and reconcile multiple ODBC data sources (and CSV files). To work, either usage demands a MySQL database modelled on my free electronic health record (***f*ehr**). The intent is simple: make and use an SQL table called **timely** that contains all of the time zone relationships in the Olson (tz) database maintained by IANA. This database is rule-based, but can with difficulty be translated into a single SQL table (with some supporting tables).

The basic concept is simple: rather than using the somewhat arcane rule-based translation of a local (wall) time within a time zone to a standardised timestamp—and back—simply use a table of transitions for the translation. In the same package we also obtain other functionality: access to the relevant SQL table, compensation for leap seconds, and a format that ticks monotonically: proleptic GPS time, which avoids the issues with leap seconds that intermittently jump back in both UTC and Unix time.

The following is all a bit quixotic, as in 2019 RFC 8536 finally documented the TZif 'time Zone Information Format', which shows how to store consistent data on universal time offsets, daylight saving (DST), and leap second adjustments for given zones. The format was originally introduced in the 1980s.[1] So if (for example) on your Linux system you say:

```
ls /usr/share/zoneinfo
```

... then you'll obtain a set of directories. To identify specific zones, try e.g.

```
ls /usr/share/zoneinfo/Pacific
```

If we now wish to pull out the actual transitions, we can say:

```
zdump -v -c 2020,2030 /usr/share/zoneinfo/Pacific/Auckland
```

You'll get something along these lines:

[1] Note that TZif uses UTC, with the attendant leap second issues.

UT timestamp	NZ timestamp	DST	Offset
Sat Apr 4 13:59:59 2020	Sun Apr 5 02:59:59 2020 NZDT	isdst=1	gmtoff=46800
Sat Apr 4 14:00:00 2020	Sun Apr 5 02:00:00 2020 NZST	isdst=0	gmtoff=43200
Sat Sep 26 13:59:59 2020	Sun Sep 27 01:59:59 2020 NZST	isdst=0	gmtoff=43200
Sat Sep 26 14:00:00 2020	Sun Sep 27 03:00:00 2020 NZDT	isdst=1	gmtoff=46800
Sat Apr 3 13:59:59 2021	Sun Apr 4 02:59:59 2021 NZDT	isdst=1	gmtoff=46800

Table 1.1: Transitions (partial)

My approach obsessively works through the original tz rules, applies them, and translates the values obtained into the **timely** table, defined as follows:

```
CREATE TABLE timely
( timekey integer
    , constraint timely_pk PRIMARY KEY(timekey)
  ,region BIGINT
    ,constraint timely_region_fk FOREIGN KEY (region)
      references PLACES(place)
  , year integer
  , transition BIGINT
  , zone_offset BIGINT
  , dst BIGINT
  , ignored integer default 0
  , ver integer default 0
  , chk int
)CHARACTER SET=utf8mb4
    COLLATE=utf8mb4_unicode_ci;SHOW WARNINGS;DESCRIBE timely;
```

The database structure is explored in detail in the document *small_time_400.lyx*, which provides full source code that includes all of the ancillary tables. But you could 'simply' populate the table by accessing each of the TZif data files, and translating the timestamps.

1.1 Some history

Initially, all of the initialisation and testing was done within the .pm module. This code was written more than a decade ago. In December 2025 I extracted command-line routines from the earlier timely_300.lyx document and moved them to *small_time_400.lyx*. The current module in *timely_400.lyx* provides just the basic functionality required.

I agonized a bit about some of the contents of this module. On the one hand, it seems clunky and just plain wrong to have a built-in (but crude) Perl command-line interface for loading and checking the importation of tz; and similarly, it seems a bit of a stretch to build the ODBC functionality and core SQL queries into this module. Surely just do one thing (importation of tz) and do it well. Conversely, there is a natural fit, it is in a sense 'complete', and building a lot of this functionality into either the front end or separate modules seems a bit crazy. It is what it is.

The following text describes the available options. The key feature that distinguishes Timely is that it provides a single SQL table of time zone transitions, in contrast to the arcane rules within tz; it also translates the latter into the former. By default Timely uses the rules it reads in to plan future transitions for all countries until 2035, based on the value in `$TOPYEAR`.

1.2 Functions

The full list of exported functions is in Section 1.6, which also lists internal functions that are not immediately accessible externally. These fall into several main categories, as shown below. As hinted above, we have SQL functionality, core time-zone functions, and some handling of errors and warnings.

1.2.1 SQL requests

The main idea is that we use ODBC to execute SQL statements, executing SQL code and retrieving the results of SQL queries.

GetSQL (4.3) Retrieve a single SQL row for a supplied query.

DoSQL (4.4) Execute an SQL statement that doesn't return a query value

SQLManySQL (4.5) Retrieve an array of SQL data—many rows

AppendSQLConstraint (4.6.1) Add another constraint (pending) to a query

ClearSQLConstraint (4.6) Remove all pending constraints

FetchManyKeys (4.2) Fetch keys (one or more) for new rows to insert.

1.2.2 Time & Timezone functions

The core Timely functions.

FindRegion (5.2) Locate a zone. This returns several values, including the first match (if present) and the number of hits. When printing to the console, submitting a '?' anywhere will result in multiple matches being printed to the console.

SetRegion (5.3) This is a clumsy legacy transformed into an important routine. Originally, it simply allowed storage within the module of both the current region name and its corresponding database code; but its current function makes more sense: load all of the database transition points for this region, and retain these as an array that will permit easy interconversion of Julian and Gregorian times for the current region.

GetLocalTime (5.4) Get local wall time into standard Gregorian format

Julian (5.5) Given a Gregorian time, convert to Julian microseconds

Gregorian (5.6) Convert a Julian time in microseconds to Gregorian. Note that the seconds are rendered as `ss.ffffff`, which may cause consternation in some databases, notably older versions of MS SQL Server.

FormatDate (5.7) Reformat a date from one of a variety of formats to standard Gregorian `YYYY-MM-DD hh:mm:ss.ffffff`

More fiddly are the following, which should be used sparsely if at all.

GpsJulian () Given Gregorian date, convert to Julian and then apply GPS adjustment.

InternalJulian () Given a Gregorian date specific to a given zone, find the internal (GPS) rendition of the date.

ModTimeNow () Return *current* seconds in the Unix Epoch, vulnerable with signed 32 bits.

JulianFromUnix () Given Unix time, convert to GPS Julian in seconds

Greg () Given Julian time in seconds, local offset and DST likewise, render Gregorian date string.

J2G () Given a speedup value (qic), a zone and a JD value using GPS time in seconds, produce an array of values needed to eventually make a Gregorian date string (adjusted for zone and DST). The array comprises Y,M,D h,m,s delta zone, delta DST, a groundhog value, as well as extra delta zone and delta DST values if a groundhog 'shadow' is present, cf. Anomalous (6.4.2).

G2J () Given speedup $qik together with $zone, $YY, $MM, $DD, $h, $m, $s, produce a Julian value in microseconds (text representation as Perl is/was unreliable here).

ToJulian () Revised Julian that takes $YY, $MM, $DD, $h, $m, $s, $fff, $Zoff, $Dst.

ApplyGps () Given that a Gregorian date has been converted correctly to the corresponding Julian day number, adjust to GPS by adding the relevant leap seconds.

HugeToJ () Convert integer microseconds (as string) to seconds.

FullGregorian () Take jd, LCL, DST and isgps (is it proleptic) return a list of year, month, day, hours, minutes and seconds.

PrettyDate () Generate Gregorian date string from $isT, $YYYY,$M,$D,$h,$m,$s, but if $isT is true, then use 'T' as separator rather than a space.

1.2.3 Errors and Warnings

Warn () Issue a warning

ListWarnings () Return text summary of the warnings

ClearWarnings () Clear the warning count

GetNewWarnings () Return the number of new warnings

TooBig Clumsy check that we haven't been forced from integer notation to exponential. If needed, return as integer string.

SetOffender () Store offending line

Eek () Force death — with documentation of the problem.

1.2.4 Configuration & servicing

Utility functions.

BeginTimely (2.1) Initialise Timely.

SetTzDatabase () Store the internal ***f*ehr** source identifier (for src field) locally.

EndTimely () Terminates certain variables (closes off prior to exit).

SetMaxKeyFetch () Maximum number of keys that can be retrieved in one go—see **FetchManyKeys** ()

ErrorThresholdGet () Get current error threshold

SetErrorThreshold () Set current error threshold

SetWarnMax () Set maximum warnings (for various levels)

SetUserId () Set ID to use as "current user"

SetParsingScript () Set description/characterisation of script currently being parsed.

XPrint () Print to console AND log file.

1.2.5 Miscellaneous

CheckOS () Check operating system and return one of Mac | Windows | Linux | Unknown (at present).

FetchKey () Generate a brand new key. Use with caution.

1.3 How tz works

This is adequately described in Wikipedia[2] but the definitive reference is in the tz source at https://www.iana.org/time-zones, notably the included document *theory.html*. An excellent overview is https://data.iana.org/time-zones/tz-how-to.html. It's important (and confusing) to realise that zone specifications designate an end date (i.e. the relevant line applies UNTIL the specified date, i.e. retrospectively) while rule specifications state the point at which a rule changes (they apply prospectively, until the next rule). Things are made more complex by the potential to specify a time as Universal Time, zone time, or wall time. The basics:

[2] https://en.wikipedia.org/wiki/Tz_database

1. A a time zone is any national region where local clocks have all agreed since 1970 (the start of UNIX time). The zones are listed in the file *zone.tab* and *zone1970.tab*; the former is simpler. This table maps country codes (2-character) to TZ zones. It also contains coordinates (North+East) that localise the TZ entry on the surface of the planet. TZ names are usually named after large cities (names of which are generally more stable than those of countries).

2. Multiple text files are named after continents and oceans: each file describes several time zones in multiple lines per zone;

3. There are some standard suffixes applied to dates:

   s local standard time = wall time *without* DST!

   g GMT

   u UTC, the same as GMT

   z nautical time zone, also = GMT

   w Wall clock time—a usage that causes a lot of pain in translation.

   If there is no suffix, this is the same as the **w** suffix

4. Comments start with `#` ; blank lines are allowed.

   (a) Comments can occur at the end of a valid line

   (b) Some lines will have whitespace (tabs etc) followed by a comment and nothing else.

5. The first line of a time zone specifies the name—continuation lines have blank name fields. Names generally are in the format AREA/LOCATION but some are further qualified e.g. America/Indiana/Petersburg. There are fairly clearly-defined rules for choosing these names, but some legacies and exceptions exist. Certain zone names also have abbreviations like EST.

   (a) The maximum length of a zone name is 14 characters, and some zone names have been modified to accommodate this. Generally a tab character will separate zone name and STDOFF, but with a sufficiently long name, the tab will be replaced by a single space. e.g. Africa/Abidjan is followed by a tab, but Atlantic/Cape_Verde by a space. Each zone line also has:

   (b) STDOFF: a GMT (UTC) offset along the lines of `0:12:12` or `1:00` (This may be negative)

   (c) RULES: The applicable rule e.g. `Algeria` ; this may also be a dash `-` or a time like `1:00`

   (d) FORMAT: The format e.g. LMT | WE%sT | CE%sT | WET | CET | -02 | -01 | WAST | +04/+05 | %s | BST | GMT/BST | MSK/MSD | EE%sT etc. The %s within some of these names is replaced by the LETTER field from the relevant rule!

   (e) [UNTIL] an end time e.g.

      i. `1882` or

ii. `1981 May` or
iii. `1977 Oct 21` or
iv. `1940 Feb 25 2:00` or
v. `1912 Jan 1 1:00u` or
vi. `1940 May 20 2:00s` ... or even
vii. `lastFri` !

This is the precise time at which the rule ends (unless it still continues to now, in which case the rule is blank

(f) An optional comment starting with #

6. The rules can be complex, with potentially multiple lines for a single rule. A rule definition line contains the name of the rule followed by (in order): NAME FROM TO TYPE IN ON AT SAVE LETTER/S

(a) FROM starting year e.g. `1958` (always \d{4})
(b) TO end year eg. `1959` OR `only` Options:
i. max
ii. 1995 etc. (always \d{4})
iii. only
(c) A TYPE, usually just a dash `-`
(d) IN : something like Mar — always a month
(e) ON: something like lastSun or perhaps Sun>=8 The options are:
i. a day of month like `1` or `14`
ii. A day of week with qualifier like `Sun>=1` or `Sat>=16`
iii. A "last" value like `lastThu` or `lastSun` (of the month)
(f) AT: A start time with two possible formats:
i. `23:00` or `7:00` (etc) ie \d\d?:\d\d The value may also be `00:00` or `24:00` (!)
ii. `23:00s` or `7:00s` Likewise for `00:00s` or `24:00s` (!)
iii. `1:00u` or `17:00u` Likewise for 00/24 as above.
(g) SAVE: The amount of daylight saving applicable, along the lines of:
i. `0`
ii. `1:00`
iii. `-1:00`
(h) LETTERS that signify various things, e.g. W for War, P for Peace. Options are S | - | M | WAT | CAT | +01 (etc i.e. +\dd) | D | W | P | MST | MMT | MDST | MSK etc.
(i) This may be followed by a #comment.

7. What about Link fields e.g. `Link Africa/Abidjan Africa/Bamako # Mali` ? This starts with "Link" and then has the LINK-FROM and LINK-TO entries.

(a) The LINK-FROM entry must be a NAME of a Zone;
(b) The LINK-TO entry is simply an alternate name (alias) for that zone.

1.4 How tz *should* work

You have just started plumbing the insane depths of tz. Here's how things might work a bit better, based on accepted time zones:

1. Have each transition for a given zone marked as a point on a standard time scale that ticks with monotonous regularity;
2. At the transition, state the rules that apply in terms of:

   Z offset from the standard (reference point: UTC or equivalent)

   DST daylight saving that applies, if present.

That should be it, but you also need rules that disambiguate (or accommodate) the gap or 'groundhog interval' that occurs when DST is applied or removed. You need to do two further things:

- Translate an invalid wall time (time in the gap) either forward or back to a valid time. You might do this with or without an offset, too.
- Signal that a wall time that falls in a groundhog interval is ambiguous—e.g. append a star, or something.

1.4.1 How does Timely work?

1. The monotonically ticking scale is GPS time, extended backwards in time—proleptically—to accommodate times before the start of January 6, 1980, with weeks beginning on Sunday.
2. The above transition details are recorded in an SQL table called **timely**, which maps each zone against a transition, z and d. For convenience, the year is recorded and a prosthetic 'hint' documents the status at the end of the year—both are convenient in calculation.

Note that at present, UTC does not tick monotonically; nor does Unix time: there are leap seconds. We need to accommodate these leap seconds in translating between GPS time and UTC.

1.4.2 How can we translate from tz to Timely?

This is explored and demonstrated in *small_time_400.lyx*. An interactive interface there creates, explores and tests. Timely is available from the command line by navigating to e.g. *C:|w|seekwell|perl|* (or ~/smalltime/perl/ — wherever you've stored the program) and saying:

```
perl small.pl
```

See *small_time_400.lyx* for details. Conversions between Julian and Gregorian should use Peter Baum's algorithms.[3] Note that the default database is called smalltime, as described in small_time_400.lyx, but you can use another target database e.g. fehr by saying something like:

```
perl small.pl db=fehr
```

There are a few other command line arguments, described in *small_time_400.lyx* too. Invocations of Timely:: are also present in the source code for vector seekwell.

[3] cf. cf. https://www.researchgate.net/publication/316558298_Date_Algorithms .

1.5 The main file

We require the Perl subdirectory in the directory in which the LyX files are stored, normally ~/smalltime/perl/ or a Windows equivalent. This must contain a *lib* subdirectory.

The file is *perl/lib/Timely.pm*; the module is Timely.pm.

```
#!/usr/local/bin/perl -w
package Timely;
use strict;
use warnings;

use feature 'unicode_strings';    # UTF-8 usage
use utf8;                         #
use open ':encoding(utf8)';       #
binmode(STDOUT, ":utf8");         #
```

1.6 Exports

We export a variety of symbols.

```
use Exporter qw(import);

our $VERSION = 4000000;          # 4.0.0 e.g. 1024000=1.24.0
our @ISA = qw(Exporter);
our @EXPORT = qw( BeginTimely );
our @EXPORT_OK = qw( GetSQL DoSQL SQLManySQL AppendSQLConstraint ClearSQLConstraint
  FindRegion SetRegion Timely GetLocalTime Julian Gregorian
  FormatDate QuickJ2G GetUtcCode SetUtcCode ExGps FetchTzDate
  GetNewWarnings Eek TooBig SetOffender SetMaxKeyFetch ClearWarnings
  EndTimely SetUserId SetTzDatabase SetParsingScript
  XPrint Warn ListWarnings FetchManyKeys ErrorThresholdGet SetErrorThreshold
  SetWarnMax CheckOS GpsJulian InternalJulian ModTimeNow JulianFromUnix
  Greg J2G Padding FetchKey ToJulian ApplyGps HugeToJ FullGregorian PrettyDate G2J
  );
  # ^ DO NOT put comment in qw list. Note Padding() is a bit niche.

## WE CURRENTLY DO *not* export the following but they are still accessible using Timely::
## ValidatePath() SetFehrOdbc()FetchLeapData() BigApplyGps()
## BumpNewWarnings() LogError() Converter() DoubleDigit() Lead()
## Greg() ClipFractionalSeconds() Anomalous() Signum() Biggen() JulianDay()
```

1.6.1 Uses

```
use Time::HiRes qw(gettimeofday usleep);  # high-resolution timing
use POSIX qw(floor);
use Term::ReadKey;  # only used to interrupt long loops by pressing Esc
             #
```

```
  ReadMode 1; # normal read mode. [should this be in invoker?]

use DateTime; # this may give trouble if you're behind a corporate firewall
              # For Windows DateTime version:  perl -M"DateTime 9999"
              #
```

DateTime documentation is at https://metacpan.org/pod/DateTime. It does not unfortunately seem to provide a method that says which version of tz is currently being used. If however you explore your local Perl installation under Linux (eg */usr/local/share/perl/* in Ubuntu) you should be able to locate a DateTime/TimeZone/ subdirectory in which there are .pm files. These should contain an Olson version number along the lines of:

```
our $VERSION = '2.66';
```

You can also say:

```
cpan -D DateTime::TimeZone
```

This will not only provide the version, but also tell you the most recent one. To update, say:

```
sudo cpanm DateTime::TimeZone
```

Under Windows, you may be a bit more crippled.

1.6.2 Constants

Some constant values.

```
use constant HUGE_INT => 999999999999999;      # if GT, has exponent in Perl
use constant HUGE_NEGATIVE_INT => -(HUGE_INT); # similarly...
use constant EPSILONSECONDS => 0.010;     # [clumsy]
use constant  PART_EMPTY => '__';  #

my $TICK  = "✔"; # or '_/' if no Utf8.
my $FETCHTIMEOUT = 2;           # seconds
my $TOPCOUNTRY = 999;           # max in PLACES table. *DO NOT* fiddle!
                                # don't make it a constant (interpolated).
```

The value of `$TOPCOUNTRY` is constant but rendered as a variable for easy interpolation. It assumes that the **PLACES** SQL table has a list of countries (See *small_time_400.lyx*) with appropriate codes < this value.

1.6.3 Clumsy variables

These are global within the module; most are global constants that should never be fiddled with. In several cases, $ variables are used rather than the clumsy Perl `use constant` specification, to permit easier interpolation in strings.

```
my $SQLCONSTRAINT = '';
my $LOGFILE;
my $ERRORFILE;
```

```
my @WARNMAX =   (200, 500, 1000, 10000); # maximum warnings displayed in each category
my @WARNCOUNT = (  0,   0,    0,     0); # actual number of warnings recorded for this run.
my $NEWWARNINGS = 0;                     # new warnings during recent stmt execution
my $ERRORTHRESHOLD = 8;                  # &XPrint_() logs statements if < threshold
my $OFFENDING_LINE;
my $MAXIMUM_KEY_FETCH = 5000000;         # can fetch 5M keys at a time (limit)

my $DB_MAIN;                             # THE HANDLE OF THE MAIN (normally, fehr) DATABASE
my $TZDATABASE;                          # a source (src, SOURCES table) ID used in src fields.
my $USERID;                              # MAIN SYSTEM 'superuser'
my $UTCCODE;                             # 'Global' code (to fill in)

my $SQLOK = 0;                           #

my $USE_DBI;                             # if 1, then are using DBI:: and not Win32::ODBC.
my $MY_DBI_HANDLE;                       # undefined, initially
my $PARSING_SCRIPT = '';                 # Identifies script currently being parsed.
```

Under Windows, to render the tick in the DOS console install font from: https://math.berkeley.edu/~serganov/ilyaz.org/software/fonts/ , in the console say: **`chcp 65001`** and then right-click on the top bar | Properties | Font | Deja vu Sans Mono unifont.

The $USE_DBI is something of a legacy for the time when Win32::ODBC was a thing. A strong case can be made for removing all of the code that depends on Win32::ODBC.

1.6.4 initial code for TZ

Time-zone conversion related.

```
## internal storage for Timely rows relevant to current zone choice:
my $MYTIMELY;
my $MYZONE = 0;          #
my @LEAPDATA;            # Stores leap second transitions.
```

2.0 Instantiate

2.1 Setup

Create a log, and get Julian timestamp. Must be called first.

```
sub BeginTimely #
{ my($dbh, $usedbi) = @_;

  &SetFehrOdbc($dbh, $usedbi);
  # $LOGFILE, $ERRORFILE are globals in this module

if (! &ValidatePath('log'))
   { die "Directory 'log' does not exist";
   };
  my($tim) = ' ' . localtime; # NB leading space (scalar context).
  $tim =~ s/[\s:]/_/mg;
  my($logfile) = "log/EDLOG$tim.txt"; ## [might wish to remove spaces?]
open ($LOGFILE, ">$logfile") or die
     "*CRASH* Could not open LOG $logfile :$!\n";

  my($errfile) = "log/ERRORS.txt";  # must *append* not overwrite!
open ($ERRORFILE, ">>$errfile") or die
     "*CRASH* Couldn't open ERROR file $errfile :$!\n";
  ClearSQLConstraint();
  @LEAPDATA = &FetchLeapData();
  return(@LEAPDATA); ## hmm, check this
}
```

Takes and establishes the database handle $dbh and whether DBI is used (`$usedbi`).

2.1.0.1 Set UTC code

Local storage of the internal database PLACES code for ‘UTC’.

```
sub SetUtcCode
{ my($utc) = @_;
  $UTCCODE = $utc;
}
```

2.1.0.2 Load leap-second-adjustment data

From the ***f*ehr** database, load leap second data from the **leapseconds** table.

```
sub FetchLeapData
{
  my($handDB) = $DB_MAIN;

  my($q) = "SELECT gpstime, utctime, toffset FROM leapseconds "
   . "order by leapsecond DESC";  # topmost is first
  my(@LEAPS) = &SQLManySQL($handDB, $q, 'get leap data');
  return(@LEAPS);
}
```

The caller will (and must) store the returned array in the global `@LEAPDATA`.

2.1.1 fehr odbc

Store the handle of the 'ODBC' connection as $DB_MAIN, and the obsolete boolean integer value for $USE_DBI.

```
sub SetFehrOdbc #
{ my($dbh, $usedbi) = @_;
  $DB_MAIN = $dbh;
  $USE_DBI = $usedbi;
}
```

2.1.2 Set parsing

Only used to represent the current script being parsed within vector seekwell. Useful in debugging (alone).

```
sub SetParsingScript #
{ my($b) = @_;
  $PARSING_SCRIPT = $b;
}
```

2.1.3 max key fetch

The maximum number of new (sequentially generated) keys that can be retrieved at one time.

```
sub SetMaxKeyFetch #
{ my($i) = @_;
  $MAXIMUM_KEY_FETCH = $i;
}
```

Exceeding this will force an error.

2.1.4 UserId

The ID of the responsible user. Used in generating temporary locks while retrieving new row keys. See usage.

```
sub SetUserId #
{ my($i) = @_;
  $USERID = $i;
}
```

2.1.5 TzDatabase

The database that contains the TZ tables.

```
sub SetTzDatabase #
{ my($i) = @_;
  $TZDATABASE = $i;
}
```

2.1.6 Close Zone

End off, closing the error and log files.

```
sub EndTimely #
{
  close $LOGFILE;
  close $ERRORFILE;
}
```

2.2 Various

2.2.1 Check OS

Check operating system:

```
sub CheckOS #
{ my $Os = 'Unknown' ;

if($^O =~ /darwin/i)
  { $Os = 'Mac';
  }
elsif($^O =~ /linux/i)
  { $Os = 'Linux';
  }
elsif($^O =~ /win/i)
  { $Os = 'Windows';
  };

  return($Os);
}
```

A crude check that distinguishes just Mac/Linux/Windows, at present. Uses the Perl `$^O` (capital O, not zero).

3.0 General subroutines

3.1 Error-related

Higher error threshold values mean that the corresponding message won't be printed, provided it's above the threshold. Set threshold.

```
sub SetErrorThreshold #
{ my($i);
    ($i)=@_;
  $ERRORTHRESHOLD = $i;
}
```

Retrieve threshold.

```
sub ErrorThresholdGet #
{ return($ERRORTHRESHOLD);
}
```

Retain offending line:

```
sub SetOffender #
{ my($offal);
    ($offal)=@_;
  $OFFENDING_LINE = $offal;
}
```

3.2 Death

Die, logging error. Can be invoked externally.

```
sub Eek #
{ my($msg, $prefix, $erk, $line);
    ($msg, $prefix, $erk, $line)=@_;
  my($ln) = '';

if(length $prefix > 0)
  { $prefix .= "\n  Warnings=" . join("\t",@WARNCOUNT) . "\n"  ;
    $prefix .= "  at Perl source line $line ";
  };
  my($out) = "$prefix$msg $ln\n";
if( (defined $OFFENDING_LINE) # SEE allocation
```

```
  &&(ref($OFFENDING_LINE) eq 'ARRAY')
  )
  { $out = "$out < @{$OFFENDING_LINE} >";
    undef($OFFENDING_LINE); # undefine so not used twice if caught!
  };
if(length $PARSING_SCRIPT > 0)
  { $out = "$out <bad script line near: '$PARSING_SCRIPT' >";
    $PARSING_SCRIPT = ''; # neat.
  };
  &LogError($out); # 'permanent' record

  # also helpful print, to accommodate debugging of EITHER [explore this]
if(  $erk
  &&($ERRORTHRESHOLD > 10)
  ) # arbitrary choice of 10 [explore]
  { my($dud) = $out;
    $dud =~ s/[\r\n]//mg;
    ## &XPrint(0, "\n*   RECOVERY <$dud>");  # clumsy
  };

  ## &XPrint(0, $out);
  die $out ;
}
```

3.2.1 Aagh

Internal to this module, a wrapper for Eek (), which is not itself otherwise invoked from within the module. Eek () takes a second, auxiliary parameter and (weirdly enough) is also only referenced once by *seek_400.lyx*, by the Aagh () routine there!

```
sub Aagh #
{ my($msg, $line, $foo);
    ($msg, $line, $foo)=@_;

  &Eek($msg, '', $foo, $line);
}
```

3.3 Path validation

```
sub ValidatePath
{ my @pth;
  my $wholep = "";
 ($_) = @_;

  if ( /^\/(.*)$/ ) # if starting slash
     { $wholep = "/";
       $_ = $1;
     };
  @pth = split /\//;
```

```
#  print "\n Debug: path components @pth";

  my $p;
  foreach $p (@pth)
    { $wholep = "$wholep$p";
      if (! (-e $wholep) )
         { # print "\n Debug: bad path $wholep";
           return 0; # fail
         };
      $wholep = "$wholep/";
    };
  return 1;
}
```

3.4 Time-related utilities

3.4.1 Modified time value

Use of `gettimeofday` in scalar context would return a single number that includes fractional seconds; for our purposes here, we simply return the seconds in the Unix Epoch.

```
sub ModTimeNow
{ my($epochseconds, $microseconds);
  ($epochseconds, $microseconds) = gettimeofday;
  $epochseconds;     # use seconds for 32-bit system [?]
}
```

3.4.2 Padding

Generate string of length $n comprising repeated character $ch.

```
sub Padding
{ my($ch, $n) = @_;
  return( $ch x int($n) ); # funny old Perl
}
```

3.4.3 DoubleDigit

```
sub DoubleDigit #
{ my($i) = @_;
  return(&Lead($i,2));
}
```

3.4.4 Leading zeroes

Given a number $i, preprend zeroes to make it up to the $target length. If longer, simply return.

```
sub Lead #
{ my($i, $target) = @_;
while(length $i < $target)
```

```
  { $i = "0$i";
  };
  return($i);
}
```

3.4.5 Sign

```
sub Sign #
{ my($n) = @_;
if($n < 0)
  { return('-');
  };
  return('+');
}
```

3.4.6 Signum

Returns 0,1 or -1 depending on sign of its single argument; 0 returns zero

```
sub Signum
{ my($n);
    ($n)=@_;
 return( $n <=> 0);
}
```

3.4.7 Biggen

If a number is just below a positive value, round up to that positive value and return an integer; similarly round *down* if just above a negative value!

```
sub Biggen #
{ my($n) = @_;
  return(  int( $n + EPSILONSECONDS*&Signum($n) )  );
}
```

3.5 Perl big issues

Any number over about `HUGE_INT` will be rendered in exponential notation. This is baad. At present we 'sacrifice the precision' (it's already sacrificed) but retain a facsimile of an integer by converting back to a string.

Subsequently, I added the $soft option, where if this is set, we don't issue a warning.

```
sub TooBig #
{ my($n, $soft) = @_;
if($n < HUGE_INT && $n > HUGE_NEGATIVE_INT)
  { return($n);
  };
if(! $soft)
  { &Warn(1, "Large integer $n. Likely precision loss");
  };
  return( int($n/1000) . '000' ); # [ugh]
}
```

3.6 Pad

```
sub Pad #
{ my($s, $L) = @_;
while(length $s < $L)
  { $s = "$s ";
  };
 return($s);
}
```

3.7 XPrint

A general purpose printing routine that may also print to the log file. Only prints if the threshold is right.

```
sub XPrint #
{ my($level, $msg) = @_;
if($level < $ERRORTHRESHOLD) # global test
  { print $msg;  # clumsy
  if($LOGFILE->opened())
    { print $LOGFILE $msg; # and keep a copy, will *fail* if no $LOGFILE [explore]
    } else
    { print "  *** BUGGER handle is closed\n\n";
    };
  };
}
```

3.7.1 Issue warning

This is more general. It counts the number of warnings issued in a given class, and calls it quits if that number has been exceeded, still however printing to the log file if indicated.

```
sub Warn #
{ my($level, $msg) = @_;
if($level > 3)
  { $level = 3;
  };
  &BumpNewWarnings($level); # short-term record first
  $WARNCOUNT[$level] ++; # bump warning count
  my($stars) = '*' x $level;  # Perl x operator
if($WARNCOUNT[$level] < $WARNMAX[$level]) # if not too many warnings:
  {
    my($W) = 'WARNING';
  if($level == 0) # minor
    { $W = 'NOTE';
    };
    &XPrint(0, "\n  $stars $W $stars $msg"); # reactivated, 2021-03-12
    return;
  } else
```

```
  { # still print to log file! [ 2021-03-13 ]
  if($level < $ERRORTHRESHOLD) # global test
    { print $LOGFILE "\n$stars $msg";
    };
  };
if($WARNCOUNT[$level] == $WARNMAX[$level])
  { &XPrint(0, "\n...suppressing level $level warnings ($WARNCOUNT[$level], too many)" );
  };
  # otherwise just ignore.
}
```

3.7.1.1 Set Warning max

```
sub SetWarnMax #
{ my($i, $K) = @_;

  $WARNMAX[$i] = $K; # might have sanity check on value [explore]
}
```

3.7.1.2 Summarise warnings

Provide an array of the warning counts.

```
sub ListWarnings #
{
  # &XPrint(0, "\nWarning counts: \n -\t*\t**\t***\n " . join("\t",@WARNCOUNT) );
  return(@WARNCOUNT);
}
```

3.7.2 Short-term warning counts

Very simple at present, but we should consider recording the changes for the different levels, and not just a global count [explore].

3.7.2.1 Clear the count

```
sub ClearWarnings #
{ $NEWWARNINGS = 0;
}
```

3.7.2.2 Bump new warning

Used internally to increase warning count.

```
sub BumpNewWarnings #
{ my($level) = @_; # unused at present
  $NEWWARNINGS ++;
}
```

3.7.2.3 Get new warnings

Note that once the *current instruction* is over, the value in `$NEWWARNINGS` should be cleared.

```
sub GetNewWarnings #
{ return($NEWWARNINGS);
}
```

3.7.3 Log but don't print

```
sub Log #
{ my($msg) = @_;
  print $LOGFILE $msg; # and keep a copy
}
```

3.7.4 Log error

Much more selective — log severe errors:

```
sub LogError #
{ my($msg) = @_;
  print $ERRORFILE "\n$VERSION : $msg"; # Keep a copy.
}
```

4.0 Database Routines

4.1 Fetch a new key

Generate a brand new key, with semaphore to prevent collisions. The auto-commit usual to MySQL has been turned off.

```
sub FetchKey
{ my($tbl, $tag) = @_;
  # NB. $USERID = global.

  my($handDB) = $DB_MAIN;  # global too.

  my($count) = 0;
while($count < 2000)  # arbitrary MAX
  { $count ++;
    my($now) = &ModTimeNow(); # seconds
    my($later) = $now + $FETCHTIMEOUT; # seconds, unless a 64-bit system
    my($q) = "UPDATE SKEYS
         SET klock = $later,
             kuser = $USERID,
             kValue = kValue + 1
         WHERE kName = '$tbl' AND klock = 0";
    my($ok) = &DoSQL($handDB, $q, $tag);  # returns number of rows affected!
```

Note that in the following SQL query, I cast the (BIGINT) result to a string.[1] This is solely to accommodate 32-bit systems, and if the user knows that the target system supports 64-bit integers *and* that a 64-bit compiler was used, then the cast can be removed.

```
  if($ok > 0)
    { my($qget) = "SELECT kValue FROM SKEYS
                WHERE kName ='$tbl'
                  AND kuser=$USERID
                  AND klock=$later";
      my($j) = &GetSQL($handDB, $qget, "GET '$tag'");
      my($qput) = "UPDATE SKEYS SET klock = 0
                WHERE kName = '$tbl'
                  AND kuser=$USERID
                  AND klock = $later";
```

[1] MySQL won't cast to varchar, so use CHAR.

```
      my($ok2) = &DoSQL($handDB, $qput, "PUT '$tag'");
    if($ok2 < 1)
      { &XPrint(0,  "Failed at reset of semaphore($tbl)\n" );  # [explore this]
      };
    if(! $j)
      { &Aagh("Oops. Bad key generation ($j), matrix $tbl ($qget)", __LINE__, 0 );
        return;
      }; # must NOT return null key.
       return($j);  # success
    } else
    { # check lock time:
      my($qbad) = "SELECT klock FROM SKEYS WHERE kName ='$tbl'";
      my($lck) = &GetSQL($handDB, $qbad, 'get lock');
    if(! $lck)
      { &Aagh("Error. Bad key on $tbl ($qbad)", __LINE__, 0 );
        return;
      };
    # NB. klock value may now be ZERO!
    if( ($lck < $now)
       &&($lck > 0)
      )    # if timed out AND not fixed
      { my($qfix) = "UPDATE SKEYS SET klock = 0
          WHERE kName = '$tbl' AND klock=$lck";
        my($ok3) = &DoSQL($handDB, $qfix, "UPDATE $tag");
      if($ok3 < 1)
        { &XPrint(0, "\nBad semaphore fix $tbl($lck)"); # [explore]
        };
        &XPrint(0, "\nFETCH *FIX* debug SUCCESS"); # [remove this line]
      };
      usleep( int(rand(10000)) ); # microsleep up to 10ms
    }; # end else
  };  # end while
  die("Error with key fetch on matrix $tbl");
}
```

4.2 Fetch multiple keys

Similar to FetchKey (), but returns `$j`, the topmost value in a sequence of keys, as requested in `$N`. It's the duty of the caller to generate keys from `$j` downwards, of length `$N`. As &FetchKey () is simply a special case of this, it should be made redundant or simply call this routine.

```
sub FetchManyKeys #
{ my($tbl, $N);
    ($tbl, $N)=@_;
  # NB. $USERID = global.

  my($handDB) = $DB_MAIN;
  my($tag) = 'fetch many';
```

```
if( ($N > $MAXIMUM_KEY_FETCH)
  ||($N < 1)
  )
  { &Aagh("Can't fetch $N keys, max: $MAXIMUM_KEY_FETCH", __LINE__, 0 );
    return;
  };

  my($count) = 0;
while($count < 200)  # arbitrary MAX number of tries
  { $count ++;
    my($now) = &ModTimeNow();
    my($later) = $now + $FETCHTIMEOUT; # seconds, unless a 64-bit system
    my($q) = "UPDATE SKEYS
         SET klock = $later,
             kuser = $USERID,
             kValue = kValue + $N
         WHERE kName = '$tbl' AND klock = 0"; # atomic
    my($ok) = &DoSQL($handDB, $q, $tag);  # returns number of rows affected!
```

Note that in the following SQL query, I cast the (BIGINT) result to a string.[2] This is solely to accommodate 32-bit systems, and if the user knows that the target system supports 64-bit integers, then the cast can be removed.

```
  if($ok > 0)
    { my($qget) = "SELECT kValue FROM SKEYS
                WHERE kName ='$tbl'
                  AND kuser=$USERID
                  AND klock=$later";
      my($j) = &GetSQL($handDB, $qget, 'get top key');
      my($qput) = "UPDATE SKEYS SET klock = 0
                WHERE kName = '$tbl'
                  AND kuser=$USERID
                  AND klock = $later"; # release semaphore
      my($ok2) = &DoSQL($handDB, $qput, 'many put');
    if($ok2 < 1)
      { &XPrint(0,  "Bad multiple key retrieval on semaphore($tbl)\n" );
      };
    if(! $j)
      { &Aagh("Oops. Bad multikey make ($j) for $tbl ($qget)", __LINE__, 0 );
        return;
      }; # must NOT return null key.
       return($j);  # success
    } else
    { # check lock time:
      my($qbad) = "SELECT klock FROM SKEYS WHERE kName ='$tbl'";
      my($lck) = &GetSQL($handDB, $qbad, 'get lock');
    if(! $lck)
      { &Aagh("Error. Bad key on $tbl ($qbad)", __LINE__, 0);
```

[2] MySQL won't cast to varchar, so use CHAR.

```
        return;
       };
     # NB. klock value may now be ZERO!
    if( ($lck < $now)
       &&($lck > 0)
      )    # if timed out AND not fixed
      { my($qfix) = "UPDATE SKEYS SET klock = 0
          WHERE kName = '$tbl' AND klock=$lck";
        my($ok3) = &DoSQL($handDB, $qfix, 'many fix');
      if($ok3 < 1)
        { &XPrint(0, "\nBad semaphore fix $tbl($lck)"); # [hmm ? die]
        };
        # &XPrint(0, "\nFETCH *FIX* debug SUCCESS"); # [? in production code]
      };
      usleep( int(rand(10000)) ); # microsleep up to 10ms
    }; # end else
  };  # end while
  &Aagh("Error with key fetch on matrix $tbl", __LINE__, 0 );
}
```

4.3 GetSQL

Given:

myODBC the ODBC handle

$Query an SQL query

$tag a label used in error messages

Return a row of data from the query. There is no ordering of data—SQL simply returns the first row encountered.

```
sub GetSQL #
{ my($myODBC, $Query, $tag) = @_;
  $SQLOK = 1;
  if (&DoSQL($myODBC, $Query, $tag) < 0)  # failed
     { # &XPrint(0, "SQL statement failed ($Query)" );
       die "SQL statement failed '$tag'";
       return ""; # fail
     };
  my(@newrow);
  @newrow = ();

if($USE_DBI)  # [explore $dbh->selectrow_array($stmt) ]
  {
  if(! defined $MY_DBI_HANDLE)
    { print "Missing DBI handle for GetSQL, '$tag'";
      undef @newrow;
    } else
    { @newrow = $MY_DBI_HANDLE->fetchrow_array();
```

```
    };
  if( (! @newrow)
    ||(scalar @newrow < 1)
    ||($MY_DBI_HANDLE->err)
    )
    { ## print "Debug GetSQL: <$Query> oops '$tag'\n";
      $SQLOK = 0;
      @newrow = ();
    };
  } else
  {
  if ( $myODBC->FetchRow() )
     { @newrow = $myODBC->Data(); #first data row
       ## print "\n SQL debug: @newrow";
     } else
     { $SQLOK = 0;
     };
  };
  return (@newrow);
}
```

4.4 Do SQL

Given:

myODBC the ODBC handle

$Query an SQL query

$tag a label used in error messages

Execute an SQL statement that *does not* return a result. The main code:

```
sub DoSQL #
{ $SQLOK=0; # default 'fail'
  my($myODBC, $Query, $tag) = @_;
  my($retcode) = 0;

if(! defined $myODBC)
  { die "Can't find handle '$tag'";
  };

if($USE_DBI)
  { #NOT generally: $retcode = $myODBC->do($Query);
    $MY_DBI_HANDLE = eval { $myODBC->prepare($Query) } or undef $MY_DBI_HANDLE;
  if(defined $MY_DBI_HANDLE)
    { $MY_DBI_HANDLE->execute() or $retcode = 1;
           ## at present don't parameterize (no placeholders/bind values)
    if($retcode) # [? also look into ->{Executed} attribute !]
      { print "Debug: DBI error in DoSQL\n";
```

```
        $retcode = $MY_DBI_HANDLE->err;  # [hmm, ? rather use $h->state]
      };
    } else
    { print "Woops! DBI cockup in DoSQL prepare stmt '$tag'.\n";
      $retcode = $myODBC->err; # pick up error code
      ## $retcode = $DBI::errstr;
    if(! defined $retcode)
      { print "No error code!";
        $retcode = 1;
      };
    };
  } else # not using DBI:
  { $retcode = $myODBC->Sql($Query);
  };

if($retcode) # if problem
  { my($sqlErrors);
  if($USE_DBI)
    { $sqlErrors = $myODBC->errstr;
    } else
    { $sqlErrors = $myODBC->Error();
    };
    &XPrint(0, "\n Oops! $sqlErrors code=$retcode ($tag): $Query");
        # problem may be exceeding _max_allowed_packet rather than "has gone away"!
  if($retcode < 1)
    { # die "Woops! (Code=$retcode)";   # [no, rely on caller to call &Aagh_() ]
      return($retcode);
    };
  if($sqlErrors !~ /\[911\].+\[1\].+\[0\]/ ) # NOT if 'no data' [911]
        ## ^ the above will need revision for DBI. Explore.
    { return(-1); # force failure
    };
    return(0);  # no data
  };

  # no problem, RETURN ROW COUNT
  $SQLOK = 1;
  if($USE_DBI)
    { $retcode = $MY_DBI_HANDLE->rows;
    } else
    { $retcode = $myODBC->RowCount();
    };
  ## &XPrint(0, "\nDebug ($Query) rows affected=$retcode");
if($retcode < 0)  # [hmm]
  { $retcode = 0;
  };

  $retcode; # rows affected [?!]
}
```

4.5 Retrieve mutiple SQL rows

Arguments are:

myODBC the ODBC handle

$Query an SQL query

$tag a label used in error messages

This returns multiple data rows, styled as a Perl array-of-arrays, with all of the associated legerdemain:

```
sub SQLManySQL #
{
  my($myODBC, $Query, $tag) = @_;

  # introduce SQL constraint of CONSTRAIN:
if(length $SQLCONSTRAINT > 0)
  { $Query .= $SQLCONSTRAINT; # append.
    # [Would not work if no 'WHERE' or is ORDER/GROUP clause]
    ## print "\n\n\n\nDEBUGGING --CONSTRAINT-- <$Query>\n\n\n";
  };
  # NO. do not say:  $SQLCONSTRAINT = ''; # once only
  #     as then _direct_ will be affected under constraint.
  #  CALLER MUST SAY: ClearSQLConstraint() when appropriate!

if( &DoSQL ( $myODBC, $Query, $tag ) < 0)
  { &Aagh("Multiple retrieval failed ($Query)", __LINE__, 0);
    return;
  };
  my(@BIGARRAY) = ();
  my(@big);

if($USE_DBI)
  {
  if(! defined $MY_DBI_HANDLE)
    { print "Missing DBI handle for SQLManySQL, '$tag'";
    } else
    { @big = $MY_DBI_HANDLE->fetchrow_array();
    while (  @big
          && scalar @big > 0
          && ! $MY_DBI_HANDLE->err )
      { push(@BIGARRAY, [ @big ]); # reference!
        @big = $MY_DBI_HANDLE->fetchrow_array();
      };
    };
  } else
  {
  while ( $myODBC->FetchRow() )     #
    { @big = $myODBC->Data();     #get data
```

```
      # @BIGARRAY = (@BIGARRAY, @big);
      # NO! make an array of arrays:
      push (@BIGARRAY, [ @big ]);   # [] makes a reference!
    };
  };
  return( @BIGARRAY);
}
```

4.6 SQL constraint manipulation

Clear any existing constraint.

```
sub ClearSQLConstraint #
{ $SQLCONSTRAINT = '';
}
```

4.6.1 Append

Multiple constrains can be ANDed together. There is currently no check that the constraint makes sense—but in any case, keep this simple and avoid silly modifications. Some common sense is required in using this function.

```
sub AppendSQLConstraint #
{ my($txt) = @_;
  $SQLCONSTRAINT .= $txt; # if length > 0, constrains an SQL retrieval (WHERE clause)
}
```

5.0 Headline routines

The following ‘headline’ routines are all externally available (exported).

5.1 Timely [REMOVED]

This functionality is now contained within *small_time_400.lyx*.

5.2 Find region

Accepts:

rgname The text name of the region e.g. Pacific/Auckland. This may be partial.

The values returned are the region code, the associated description (like ‘Pacific/Auckland’) and the number of matches, relevant if a partial string was matched. The first value encountered in the database is returned as code and description.

The database handle is obtained from the global ***f*ehr** handle $DB_MAIN. The addition of `AND reason > -1` allows suppression of obsolete places. A further amendment allows `$rgname` to be the region number (as text), in which case description is returned with that region, if found!

```
sub FindRegion #
{ my($rgname) = @_;

  my($handDB) = $DB_MAIN;

  my($q);
  my($rgn, $desc) = (0, '');
  my($hits);
  my($showall) = 0;
  my($exact) = 0;

if($rgname =~ /\?/ )  # contained ? means 'search and then show all names!'
  { $showall = 1;
    $rgname =~ s/\?//g; # remove the '?'s
  };

  # use single or double quotes for exact match: e.g. for 'MST' EST EET and MET
if($rgname =~ /['"]/ )  # contained ? means 'search and then show all names!'
```

```
  { $exact = 1;
    $rgname =~ s/['"]//g; # remove the quotes [imprecise but ok]
  };

if($rgname =~ /^\s*(\d+)\s*$/ )
  { $q = "SELECT place, description FROM PLACES "
         . "WHERE p_amended BETWEEN 1 and $TOPCOUNTRY "
         . "AND reason > -1 "
         . "AND place = $1";
  }
elsif( $rgname =~ /^\s*([\w\/]+)\s*$/ )
  { $rgname = $1;
  if($exact)
    { $q = "SELECT place, description FROM PLACES "
         . "WHERE p_amended BETWEEN 1 and $TOPCOUNTRY "
         . "AND reason > -1 "
         . "AND description = '$rgname'"; # exact match!
    } else
    { $q = "SELECT place, description FROM PLACES "
         . "WHERE p_amended BETWEEN 1 and $TOPCOUNTRY "
         . "AND reason > -1 "
         . "AND description LIKE '\%$rgname\%' ORDER BY description"; #incl 1 = UTC
    };
  }
else
  { print "Bad search string '$rgname'\n"
        . "Name can contain a-z A-Z _ and /\n";
    return(0,'',0);
  };

  my(@M) = &SQLManySQL($handDB, $q, 'find multiple name matches');
  $hits = scalar @M;
if($hits)
  { ($rgn, $desc) = ($M[0][0], $M[0][1]); # only use first row values
  if($showall)
    { my($str) = '';
      my($i) = 0;
    while($i < $hits)
      { $str .= $M[$i][1] . ' ';
        $i ++;
      };
      print "Matches: $str\n";
    };
  };
  return($rgn, $desc, $hits);
}
```

5.3 Set region

From a primitive storage routine, this is now more sophisticated, from v 2. We:

1. Validate the region code and name;
2. retrieve all transition times from the **timely** table, in order, and store these within `$MYTIMELY`.

For usage of `$MYTIMELY`, see QuickBetween ().

```
sub SetRegion #
{ my($rgn) = @_;

  # first, deal with UTC:
if($rgn == $UTCCODE)
  { undef($MYTIMELY); # <gosh>
    $MYZONE = $rgn;
    return;
  };

  my($handDB) = $DB_MAIN;
```

Retrieve and save all **timely** rows for this region:

```
  my ($qa) = "SELECT year, transition, dst, zone_offset, ignored "
           . "FROM timely WHERE region = $rgn ORDER BY transition";
  my(@TM) = &SQLManySQL( $handDB, $qa, "fetch region ($rgn) details"); #
if(scalar @TM < 1)
  { &Aagh("Failed to retrieve region data for $rgn", __LINE__, 0);   # severe.
    return;
  };

  $MYTIMELY = \@TM; # [? make this less clunky][fix me!]
  $MYZONE = $rgn;
}
```

5.3.1 GetUtcCode

```
sub GetUtcCode #
{ return($UTCCODE);
}
```

5.3.2 Quick between

Given low and high years, quickly retrieves all rows that match, from `$MYTIMELY`, paring them down to the following columns, in order:

transition, dst, zone_offset, ignored.

Note that `$MYTIMELY` is already sorted by SetRegion (5.3) in ascending order based on timekey, and that the only additional column is **year**. So the source columns with their indices are:

0 year

1 transition, proleptic Julian GPS microseconds

2 dst, microseconds

3 zone_offset, microseconds

4 ignored (0/1)

If there is no match, an empty array will be returned.

```
sub QuickBetween #
{ my($lowY, $hiY) = @_;

  my(@TM) = @{$MYTIMELY}; # dereference [check efficiency ??][explore]
  my(@OUT) = ();

  my($L) = scalar @TM;
  ## print "Debug: number of rows in QuickBetween_() is $L\n";

  my($i) = 0;
  my($thisY) = 0;
  # this can be made faster by using a binary search to find low value.
while( ($thisY <= $hiY)
     &&($i < $L)
     )
  { my(@LN) = @{ $TM[$i] }; # get row
    $thisY = $LN[0];
    ## print "[check: $thisY $lowY $hiY]";
  if( ($thisY >= $lowY)
    &&($thisY <= $hiY)
    )
    { my(@F) = @LN[1..4];
      ## print "ok! @F \n";
      push(@OUT, \@F); # push the reference, ordered. Can't say \@LN[1..4] BTW.
    };
    $i ++;
  };

  return(@OUT);
}
```

5.4 Local time

Takes no arguments, but returns the current local (wall) time, retrieved from the system. The format is YYYY-MM-DD hh:mm:ss, as a string. The year is four decimal digits long, and if any other numeric value has under 2 digits, it's left-padded with a zero, for example 0 is rendered as "00".

```
sub GetLocalTime #
{ my($sec, $min, $hour,
```

```
      $mday, $mon, $year,
      $wday, $yday, $isdst);
  ($sec, $min, $hour,
      $mday, $mon, $year,
      $wday, $yday, $isdst) = CORE::localtime(time);
  $year += 1900;      #fix y2k :)
  $sec  = &DoubleDigit($sec );
  $min  = &DoubleDigit($min );
  $hour = &DoubleDigit($hour);
  $mday = &DoubleDigit($mday);
  $mon  = &DoubleDigit($mon );
  $mon ++;            #january is zero!
  return ("$year-$mon-$mday $hour:$min:$sec");
}
```

5.5 Julian

Convert local, Gregorian time to Julian microseconds. Accepts:

$handDB The fehr database handle

$d a string in the format YYYY-MM-DD hh:mm:ss.ffffff

The returned value is a big integer in microseconds after the start of Julian time, as proleptic GPS time rather than UTC, because the former ticks monotonically. Note that as fractional seconds do not depend on Julian or Gregorian time, they should simply be retained. There is also an issue of what to do with the second value returned, the “shadow”, which is:

0 for normal values;

-ve if the date supplied was invalid because it fell into a ‘limbo’ period where daylight saving or zone time jumped forward. The size of the negative number indicates the duration of limbo;

+**ve** if there is a second, equally valid Julian time because daylight saving or zone has jumped back—a ‘groundhog’ hour. In this case (unlike tz) the first time is always returned, and the second can be obtained by adding the shadow value.

```
sub Julian #
{ my($d) = @_;

if(! $MYZONE)  # 0 is taken as an error
  { &Aagh("For Julian_() you must first choose a region", __LINE__, 0);
    return;
  };
  my($zone) = $MYZONE;

if(!defined $d)
  { return(0);
  };
```

```
if($d !~ /(\d{4})-(\d{2})-(\d{2})[ T](\d{2}):(\d{2}):(\d{2})\.?(\d*)/ )  # strict
  { return(0);
  };
  my($yyyy, $mm, $dd, $h, $m, $s, $ffffff) = ($1, $2, $3, $4, $5, $6, $7);

  my($JD, $shadow) = &G2J(1, $zone, $yyyy, $mm, $dd, $h, $m, $s);

  # fix up the float:
  my($micros) = &MicroSecs($ffffff); # not fractional; truncated not rounded.

  return( $JD . $micros, $shadow );
}
```

5.5.1 MicroSecs

Given text digits “after the decimal point”, convert to microseconds. Should always be six digits, right-padded if necessary, so for example “5” becomes 500000; “003” becomes “003000”; “1234575” becomes “123457”. Truncates fractional microseconds, doesn’t round.

```
sub MicroSecs #
{ my($d) = @_;

  my($lF) = length $d;
if($lF < 6) # will also deal with ''
  { $d .= &Padding( '0', (6-$lF) );  # we want 6 digits, even if zeroes.
  }
elsif($lF > 6)
  { $d =~ substr($d, 0, 6); # truncate!
  };
  return($d);
}
```

5.6 Gregorian

Given a Julian value in microseconds, turn into a Gregorian date for the current zone, with fractional seconds too.

```
sub Gregorian #
{ my($JD) = @_;

  # zone:
  my($zone) = $MYZONE; # global
if(! $zone)
  { &Aagh("Gregorian_() lacks zone", __LINE__, 0);
    return;
  };

  # convert microseconds to seconds.
  my($ffffff);
```

```
if( $JD !~ /^(\d+)(\d{6})$/ )
  { return(''); # how do we fail ?
  };

  $JD = $1; # J2G_() requires seconds
  $ffffff = $2;

  my($YYYY,$MM,$DD, $h,$m,$s, $dz, $dst, $hog, $deltz, $delds)
     = &J2G(1, $zone, $JD);
  return( &PrettyDate(0,$YYYY,$MM,$DD, $h,$m,$s) . ".$ffffff" );
}
```

5.7 Rejig Date

If the date format is aberrant, try to fix this.

> <YYYYMMDD> means split up and add 00:00:00, similar for
> <YYYY-MM-DD> etc.

We also now permit:

> <YYYY-MM-DD hh:mm>
> <YYYY-MM-DD hh:mm:ss>
> <DD/MM/YYYY hh:mm>
> <DD/MM/YYYY hh:mm:ss>

Passing the entire string every time is inefficient, and a code might be used instead. Might even make a template regex where we intercalate \d for each Y M D (or even h m s).

```
sub FormatDate
{ my($template, $vlu) = @_;
  my($YY,$MM,$DD);
  my($h,$m,$s) = ('00','00','00');
if($template eq 'YYYYMMDD')
  {
  if($vlu !~ /(\d{4})(\d{2})(\d{2})/)
    { return('');
    };
    $YY = $1;
    $MM = $2;
    $DD = $3;
  }
  # might insert variants on the above here, if required
elsif($template eq 'MM-DD-YYYY')
  {
  if($vlu !~ /(\d{2})[.\/-](\d{2})[.\/-](\d{4})/ )
    { return('');
    };
    $YY = $3;
```

```
    $MM = $1;
    $DD = $2;
  }
elsif($template eq 'DD-MM-YYYY')
  {
  if($vlu !~ /(\d{2})[.\/-](\d{2})[.\/-](\d{4})/)
    { return('');
    };
    $YY = $3;
    $MM = $2;
    $DD = $1;
  }
elsif($template eq 'YYYY-MM-DD')
  {
  if($vlu !~ /(\d{4})[.\/-](\d{2})[.\/-](\d{2})/)
    { return('');
    };
    $YY = $1;
    $MM = $2;
    $DD = $3;
  }
```

Further formats. Note that although we *must* use the - separator in the *specification*, the actual text can use '.', '-' or '/'. We ignore fractional seconds.

```
elsif($template eq 'YYYY-MM-DD hh:mm:ss')
  {
  if($vlu !~ /(\d{4})[.\/-](\d{2})[.\/-](\d{2}) (\d{2}):(\d{2}):(\d{2})/)
    { return('');
    };
    $YY = $1;
    $MM = $2;
    $DD = $3;
    $h = $4;
    $m = $5;
    $s = $6;
  }
```

no seconds:

```
elsif($template eq 'YYYY-MM-DD hh:mm')
  {
  if($vlu !~ /(\d{4})[.\/-](\d{2})[.\/-](\d{2}) (\d{2}):(\d{2})/)
    { return('');
    };
    $YY = $1;
    $MM = $2;
    $DD = $3;
    $h = $4;
    $m = $5;
  }
```

UK/NZ dates:

```
elsif($template eq 'DD-MM-YYYY hh:mm:ss')
  {
  if($vlu !~ /(\d{2})[.\/-](\d{2})[.\/-](\d{4}) (\d{2}):(\d{2}):(\d{2})/)
    { return('');
    };
    $DD = $1;
    $MM = $2;
    $YY = $3;
    $h = $4;
    $m = $5;
    $s = $6;
  }
```

UK without seconds:

```
elsif($template eq 'DD-MM-YYYY hh:mm')
  {
  if($vlu !~ /(\d{2})[.\/-](\d{2})[.\/-](\d{4}) (\d{2}):(\d{2})/)
    { return('');
    };
    $DD = $1;
    $MM = $2;
    $YY = $3;
    $h = $4;
    $m = $5;
  }
```

US:

```
elsif($template eq 'MM-DD-YYYY hh:mm:ss')
  {
  if($vlu !~ /(\d{2})[.\/-](\d{2})[.\/-](\d{4}) (\d{2}):(\d{2}):(\d{2})/)
    { return('');
    };
    $MM = $1;
    $DD = $2;
    $YY = $3;
    $h = $4;
    $m = $5;
    $s = $6;
  }
```

US without seconds:

```
elsif($template eq 'MM-DD-YYYY hh:mm')
  {
  if($vlu !~ /(\d{2})[.\/-](\d{2})[.\/-](\d{4}) (\d{2}):(\d{2})/)
    { return('');
    };
```

```
    $MM = $1;
    $DD = $2;
    $YY = $3;
    $h = $4;
    $m = $5;
  }
```

... or fail:

```
else
  { &Aagh("Unknown _julian_ date format option <$template>", __LINE__, 0);
    return;  # need smarter approach here, more options
  };
  return("$YY-$MM-$DD $h:$m:$s");
}
```

6.0 Core code

6.1 Gregorian to Julian

There are some catches in turning a 'conventional' Gregorian date to a Julian value:

1. Some apparently valid, DST-adjusted timestamps may not exist (There's a skipped interval at the start of DST; this is also potentially a problem where there's a transition in the zone offset);
2. There's a "groundhog hour" (or partial hour) at the end of DST, so a supplied timestamp may have two possible 'candidate' Julian timestamps.
3. The conversion of the apparent Julian timestamp depends on GPS reference times, but in order to make this determination, you logically need to convert to GPS first!

We accommodate these as follows:

1. Determine the year and zone, retrieve the relevant database row. As the year always refers to Gregorian wall time, this will be the correct row.
2. Convert the Gregorian date G to a Julian day number K. This date however may contain a non-zero zone offset, and a non-zero daylight saving time value. We need to get rid of these:
3. Create M = K - zone_offset, also clear the transition flag T = 0;
4. If **zone_transition** is not zero, calculate N = K - zone_future:
   (a) If M and N are both < zone_transition, use M;
   (b) else, if M and N are both >= zone_transition, set M = N and T =1;
   (c) else, they are discrepant.
      i. If M > zone transition then N must be less. This implies that at the transition, the Gregorian (wall) time skipped back, so we have a 'groundhog' period. Retain M, but set T =1; [CHECK THE LOGIC HERE]
      ii. else, there must be have been a gap when we changed from zone_offset to zone_future, in other words, the Gregorian wall time represents an error. Signal this [Can we be sure that the specified time is in the gap? CHECK ME!]

(d) [THERE IS STILL A POTENTIAL PROBLEM WITH DST influencing the above. EXPLORE]

5. Determine the existing DST offset D. Initially assume there's no DST active, so set D = 0:

   (a) if **dst_start** and **dst_end** are both zero, this D=0 assumption is valid;

   (b) else if **dst_start** is zero, by (a) dst_end is non-zero, and we have carry over from the previous year to terminate DST at **dst_end**:

      i. if M >= **dst_end**+**prior_add**, then no DST (We add prior_add, as anything between dst_end and dst_end+prior add must be in the groundhog hour, so we just assume DST)

      ii. else D = **prior_add**.

   (c) If **dst_end** is zero, then we begin DST in this year, and continue to the next:

      i. if M >= dst_start, then we are in DST, and D = dst_add;

         A. else if M >= dst_start-dst_add, then report an error (no-go);

         B. else leave D = 0.

   (d) else if **dst_start** < **dst_end** we're in the northern hemisphere:

      i. if M < **dst_start**, it is possible that M is in the "no-go" area which can't exist as DST would have been added:

         A. if M is also < (dst_start - dst_add), then leave D=0, we're not in daylight saving;

         B. Otherwise, report an error, as this time doesn't exist!

      ii. if M >= **dst_end:**

         A. If M is also >= (dst_end + dst_add), then leave D = 0, as we're definitely not in daylight saving;

         B. Otherwise, there are two candidate Julian times that correspond to our Gregorian-esque value. Assume the earlier, i.e. D = **dst_add**. Practically, we can unite A. and B. into a test for M >= dst_end+dst_add, failing which D = dst_add.

      iii. else D = **dst_add**;

   (e) else we are in the S hemisphere:

      i. if M < dst_end, we are in DST from the preceding year, and D = **prior_add**;

         A. if M < dst_end + prior_add, there are two candidates, as above. Assume D = **prior_add**. Practically, we can once more unite A and B into: if M < dst_end+prior_add, D=prior_add;

      ii. else if M >= dst_start+dst_add, we are in DST starting in this year. D = **dst_add**;

         A. if however M >= dst_start, report an error, as this is a no-go time!

iii. Otherwise, leave D = 0.

6. Adjust M by *subtracting* D (which may be zero);
7. also adjust for leap seconds using the relevant UTC (leap-second-adjusted) leap time cutoff, calling this J.
8. If zone_transition is nonzero, re-check J against this value.
   (a) If T == 0 and J < zone transition, ok;
   (b) If T == 1 and J >= zone_transition, ok;
   (c) Otherwise [EXPLORE THE PROBLEM, this is to accommodate the possibility that our interpretation of the cutoff was skewed by failing to accommodate DST. BUT NOTE that the DST criteria themselves may in turn have been affected by an erroneous value related to zone_transition. A mess. [fix me!]]

6.1.1 G2J

Using the internal ***f*ehr** database handle, take the supplied internal zone ID and Gregorian date (as components) and return two things: the corresponding Julian date, and a "Shadow" value, ordinarily zero, unless we're dealing with a "special case", of which there are two:

1. Because there is a gap (zone of limbo) after any point where delta T (ΔT), the sum of the change in zone offset (Z) and the daylight saving time (D), is over zero, some supplied Gregorian timestamps can be invalid.
2. There is a re-run of time (groundhog zone) around the time of a backwards transition—where the sum of the above changes is less than zero. Therefore, one Gregorian timestamp will here correspond to two Julian times. Conventionally, tz software always returns the later of these two, when asked to discriminate.

I thus:

1. Signal a limbo time (error) by returning the imputed Julian time accompanied by a *negative* ΔT as the shadow value.
2. Signal a groundhog time by returning the *earlier* matching Julian time (not the later one, as tz software does), as well as a positive value in ΔT, the returned shadow value. This allows resynthesis of both values.

In more detail:

1. Convert the Gregorian YYYY-MM-DD h:m:s to a corresponding UTC timestamp, expressed as Julian microseconds (not yet as GPS time)
   (a) Call this converted timestamp "JBIG".
   (b) If the desired zone is "UTC", simply return JBIG after converting to proleptic Julian GPS time a seconds, and adjusting for GPS.

2. For the relevant zone, from the database retrieve all data for the years "around" the desired year. These data extend from the start of the previous year, to two years in the future, up to the last timestamp in December of that year.

   (a) Note that in the **timely** table, we will always have at least one apparent transition per year, even if there is no formal transition, as one will have been intercalated for December 31 at 23:59:59, local (wall) time. Therefore there will be at least one timestamp prior to the current time, and because of the above sampling, there will be at least two for the period after the end of the target year.

   (b) All Z (zone offset) and DST (daylight saving) values in the database are signed, in microseconds. Each pair of values for a given row applies prior to the transition time (let's call this "tr"), from the preceding transition.

3. Work upwards, examining each transition time in the database until JBIG is greater than tr. The logical approach is to compare tr with the JBIG, adjusted as follows:

$$J_{\mathrm{BIG}} - (Z_{\mathrm{tr}} + \mathrm{DST}_{\mathrm{tr}}) < \mathrm{tr}$$

   (a) This removes the influence of these variables on the wall time, and should give us the corresponding UTC time. So with this adjustment, if the test holds, we have located the interval into which JBIG falls.

4. This naive approach is however problematic because:

   (a) Each transition time is in proleptic GPS Julian microseconds, but JBIG is not yet adjusted in the minor way needed for GPS; this is easily done.

   (b) The very fact that we use Z and DST implies that these are assumed to be valid—but the values may change the interpretation of the result!

   (c) Transitions may follow rapidly after one another, even if we exclude the added complexity of the intercalated end-of-year transition.

5. The following observations are thus relevant. First, assume that the adjusted timestamp is valid. Then check this assumption. There appear to be several not-completely-exclusive possibilities:

   (a) The *preceding* interval was correct. The problem with this option is that the antecedent calculation (we have that year at the start) did not show this to be the case; our refutation is wrong.

   (b) The current *is* correct. This is quite possible, and does not exclude the possibility that there is a second valid wall time after the transition. For example, if daylight saving ends on 27 September at 02:00 (wall time), then two Julian dates will correspond

to a wall time of 01:30 on that day, one before the transition, and one after.

i. There is also the fairly trivial case where $(Z' + \text{DST}') = (Z + \text{DST})$. There are two sub-cases here:

A. The change in Z balances the change in DST. Time ticks monotonously, the adjustment is valid.

B. We're in the interval after a "December prosthetic" transition (Ordinarily there shouldn't be a proper "transition" if no change occurred).

(c) A future timestamp may be correct but our calculated one isn't, in other words, the actual timestamp has been inappropriately "pulled back" before tr when we added Z and DST. Logically, this refutation can only have happened if we're close to tr, the sum of Z and DST is positive—so that subtraction of this sum yanks things back before the transition—*and* the sum of Z' and DST' (the adjustments for the *next* interval, tr') is less than the prior sum. As this all seems a bit speculative, consider the case where the "proper" timestamp is 1 minute after tr, and Z+DST for tr is 60 minutes. This will indeed pull JBIG back before tr, as the positive value is subtracted. Next, assume that Z'+DST' is say, zero. But this implies that we have a "groundhog hour", so there will be two timestamps corresponding to the wall time, and tr will be pulled back, refuting the assumption that our calculated timestamp is invalid. This applies more generally than our example, whenever $(Z' + \text{DST}') < (Z + \text{DST})$.

(d) Finally, what of the case where our wall time is invalid—it falls into a gap? This can only occur if there's a gap, in other words $(Z' + \text{DST}') > (Z + \text{DST})$. Let's assume that the spurious JBIG is invalid because it refers to a non-existent time "in limbo". A simple example is where Z is zero (or unchanged) and we start summer time, DST going from 0 to 60 minutes, at 02:00. A wall time of 02:01 is in limbo. Now how is this value treated before and after the transition? In the antecedent period, were the time valid, the usual $J_{\text{BIG}} - (Z_{\text{tr}} + \text{DST}_{\text{tr}})$ adjustment would have put the time before tr, but with the same approach, the invalid time will appear to be after tr. However, in the new interval for tr', because of the above inequality, under the new rules the time will be moved *before* the transition, and also be invalid!

6. A further problem might be where one transition rapidly follows another:

7. From the above logic, this should work:

(a) Work upwards, for each interval adjusting JBIG by subtracting $(Z + \text{DST})$. Call this adjusted value JX.

(b) If this value is < tr, we have a potential winner:

i. If however the value is < the antecedent tr, the time is in limbo: signal this failure by returning JX and a *negative* second value (a 'Negative Shadow', the magnitude of which is the size of the gap, i.e. $(Z + \mathrm{DST}) - \left(Z^{-1} + \mathrm{DST}^{-1}\right)$ which will be negative).
ii. Otherwise, determine Delta, i.e. $(Z + \mathrm{DST}) - (Z' + \mathrm{DST}')$ and add this 'Shadow' value to JX. If this sum exceeds tr, then we have a groundhog value. Signal this by returning the unadjusted JX and the (positive) Shadow value.
iii. If none of the above applies, return JX and a Shadow of zero.

Minor factors are the use of and adjustment for GPS time, and the fact that we wish to return both of our results as (Julian) seconds, not microseconds.

```
sub G2J #
{ my($qik, $zone, $YY, $MM, $DD, $h, $m, $s) = @_; # no fractional seconds

  my($handDB) = $DB_MAIN;

if( $qik
  &&($zone != $MYZONE) # check it's the right region, D'Oh!
  )
  { &Aagh("Can't do quick retrieval, mismatched zones: $zone|$MYZONE", __LINE__, 0);
    return;
  };
```

Retrieve the relevant database values into the array `@ROWS`. Formerly, we used to laboriously perform an SQL query; now we simply use the value in `$MYTIMELY`, instantiated by SetRegion ().

```
  my(@ROWS);
  my($hi) = 2 + int($YY);  # high year limit
  my($lo) =-1 + int($YY);  # low year
if($qik)  # $zone is redundant
  { (@ROWS) = &QuickBetween($lo, $hi); # get tr, dst, zo, ignored from $MYTIMELY
  } else
  { my($q) = "SELECT transition, dst, zone_offset, ignored from timely "
      . "WHERE region = $zone AND year BETWEEN $lo AND $hi ORDER BY transition"; # ASC
     # ENCOMPASSES year of interest!
    (@ROWS) = &SQLManySQL($handDB, $q, 'get all rows');
  };
```

Initial basics:

```
  my($jraw) = &ToJulian($YY, $MM, $DD, $h, $m, $s, 0, 0, 0); # no zone/DST yet
  my($JBIG) = $jraw * 1000000; # microseconds, as in database.

  my($JX) = 0;        # if no row applies, will fail, returning zero
  my($shadow) = 0;
```

```
  my($last_tr) = 0;   # prior transition as error trigger
  my($lastd) = 0;     # dual usage as below
  my($lastzo) = 0;

if($zone == $UTCCODE)
  { $JX = &BigApplyGps($JBIG);  # GPS correction!
    return( &HugeToJ($JX), 0 );  # return GPS-adjusted Julian value, no shadow.
          # ^ [TEMP]
  };
```

Work through the rows:

```
while(scalar @ROWS > 0)     # [clumsy]
  { my($r) = shift @ROWS;
    my($tr, $d, $z, $ign) = @$r;  # transition, DST, Z, all in microseconds.
    $JX  = &BigApplyGps($JBIG - $z - $d);  # final GPS correction!
```

We have a 'winner', but first test for limbo:

```
  if($JX < $tr)
    {
      # is this in limbo?
    if($JX < $last_tr)
      { $shadow = ( ($lastzo + $lastd) - ($z + $d) ) / 1000000; # negative, seconds
   #     print "limbo warning: $JX , $shadow\n"; # debug
        return( &HugeToJ($JX), &Biggen($shadow) );  # return GPS-adjusted Julian INT
      };
```

Next, do we have a groundhog value? Note the use of >= rather than > in the following, to accommodate the "start of the hour before the transition".

```
      $r = shift @ROWS;  # get NEXT entry
      my($next_tr, $next_d, $next_z, $next_ign) = @$r;

      my($Delta) = ($z + $d) - ($next_d + $next_z);
    if($JX + $Delta >= $tr)  # yes, groundhog:
      { $shadow = $Delta/1000000; # [assume integer] [hmm]
  #     print "groundhog note: $JX, $shadow\n"; # debug.
        return( &HugeToJ($JX), $shadow);
      };
```

Otherwise, just a 'winner':

```
      return( &HugeToJ($JX), 0 );  # return GPS-adjusted Julian value INTEGER.
    }; # end winner.
```

Not a winner, continue:

```
    $last_tr = $tr;  # retain
    $lastd = $d;
    $lastzo = $z;
  };     # end of while loop.
```

Terminate. As this should never happen, signal failure by sending back a transition value of zero.

```
  return(0,0);
}
```

We also need to consider the scenario where a “transition” in tr is just a December prosthesis. This shouldn’t be an issue where JX falls before tr, as the value is “true”, and the same should generally hold if the subsequent transition is prosthetic, as it then merely represents the future state in a valid way. But what if there’s an unreasonably short gap between a prosthetic and either an antecedent or subsequent transition?

6.2 Julian

Revised version that works across dates; $ff is milliseconds, or can be left as a null string. $Zoff is the time zone offset, and $Dst is daylight saving adjustment.

NOTE that the units of the zone offset and daylight saving are *hours*, something that may cause confusion.

The algorithm is Baum’s (See Section 6.3 for a reference).

1. Z = Y + int((M-14)/12);
2. IF M<3 THEN M=M+12; F = int((153*M-457)/5);
3. J=D+F+365*Z+floor(Z/4)-floor(Z/100)+floor(Z/400) + 1721118.5

The submitted values in $Zoff and $Dst are in seconds. The returned Julian value is in seconds, not Julian day number.

```
sub ToJulian
{ my($YY, $MM, $DD, $h, $m, $s, $fff, $Zoff, $Dst);
    ($YY, $MM, $DD, $h, $m, $s, $fff, $Zoff, $Dst)=@_;
if(! defined $s) # must have seconds
  { return(0);
  };

  $Zoff /= 86400;
  $Dst /= 86400;

  $fff = &Lead($fff,3); # leading zeroes for milliseconds [?? check this]

  ## my($Z) = $YY + int( ($MM-14)/12 ); # step 1.
if($MM < 3)                          # step 2.
  { $MM += 12; # could equally just subtract 1 from $YY and use ipo $Z.
    $YY -= 1;  # equivalent of step 1.
  };

  my($F) = int( (153*$MM - 457)/5 );
  my($J) = $DD + $F + 365*$YY + floor($YY/4) - floor($YY/100)
            + floor($YY/400) + 1721118.5; # step 3.
```

```
  $J -= ( $Zoff + $Dst ); # fractions of a day
  $J += (  $h + ( $m + ($s + "0.$fff")/60 )/60  )/24;

  return($J*86400);
}
```

In the above, we take a plain Gregorian UTC date and convert it to the corresponding Julian day (floating point). We do not offset the result with a GPS offset, something done by GpsJulian () below.

6.2.1 GPS Julian

Given Gregorian date, convert to Julian and then apply GPS adjustment. Note that the values in $LToff and $Dst are in seconds, so the caller must be aware of this. Returns a value in *seconds*, not a Julian day number.

```
sub GpsJulian #
{ my($fy, $fm, $fd, $fh, $fmi, $fs, $ff, $LToff, $Dst);    # clumsy, use @
    ($fy, $fm, $fd, $fh, $fmi, $fs, $ff, $LToff, $Dst)=@_;

  my($J) = &ToJulian($fy, $fm, $fd, $fh, $fmi, $fs, $ff, $LToff, $Dst);
                                                # seconds ^       ^
  return( &ApplyGps($J) );
}
```

6.2.1.1 Add GPS

Given that a Gregorian date has been converted correctly to the corresponding Julian day number, adjust to GPS by adding the relevant leap seconds. The supplied value must be in Julian seconds, not days!

```
sub ApplyGps #
{ my($J) = @_;

  my($Ltop) = scalar @LEAPDATA;
  my($i) = 0;
while($i < $Ltop)
  { my($d) = $LEAPDATA[$i]; # most recent value is at start of @LEAPDATA
    my(@D) = @$d; # dereference
    # in the following note 1 (not 0) index:
  if( $J >= ($D[1]/1000000.0) ) # smarter ? pre-scale @LEAPDATA or a copy
    { return($J+$D[2]);  # add the offset
    };
    $i ++;
  };
  return($J-9);
}
```

6.2.1.2 GPS applied to BIG value

Adjust a big Julian (microseconds as opposed to days).

```
sub BigApplyGps #
{ my($Jbig);
    ($Jbig)=@_;
  my($Ltop) = scalar @LEAPDATA;
  my($i) = 0;
while($i < $Ltop)
  { my($d) = $LEAPDATA[$i];
    my(@D) = @$d; # dereference
    my($greg) = $D[1];
    my($off) = $D[2];    # seconds
  if($Jbig >= $greg)        # [check this inequality for consistency ??]
    { my($K) = $Jbig+($off*1000000);  # microseconds
      return($K); # days
    };
    $i ++;
  };
  my($K) = $Jbig -(9000000);  # 9s
  return($K);
}
```

6.2.2 Julian from Unix

Given Unix time, convert to GPS Julian (not simply UTC Julian). Unix time zero in GPS is 1 January 1970 00:00:00. The returned Julian value is in seconds, not days.

```
sub JulianFromUnix
{ my($unix) = @_;
  # currently allow a proleptic negative value [explore]

  my($Uzero) = &ToJulian(1970, 1, 1, 0, 0, 0, 0, 0, 0); # to seconds
  $Uzero += $unix; # [NB. what about internal unix leap seconds?]
  return( &ApplyGps($Uzero) );
}
```

6.2.3 Fetch TZ date

Given year, month etc, establish a date for the stated zone (eg “Pacific/Auckland”) and then convert to UTC, returning the string value. This is problematic in that some local timestamps simply won’t exist. We thus always provide $JD, which is a GPS Julian time: it can be converted to UTC, and the corresponding date/time extracted.

In other words, our sequence is:

1. Submit a Gregorian wall timestamp (together with a backup Julian timestamp);
2. Turn this into a value (object) within DateTime using the zone name in `$zone`; DateTime uses tz (This may fail)
3. On success, clone the value, and convert it to a UTC timestamp, using DateTime again

(a) Return the formatted string.

4. On failure, convert the Julian value to UTC, convert *this* to a UTC timestamp, and return the formatted string. A compensatory hack.

(a) Write some warning messages.

Note that when tz converts an ambiguous wall time to UTC, it returns the later of the two UTC dates.

```
sub FetchTzDate #
{
  my($JD, $zone, $YEAR, $MONTH, $DAY, $HOUR, $MINUTE, $SECOND) = @_;
##  print " { FetchTzDate : $zone, $YEAR, $MONTH, $DAY, $HOUR, $MINUTE, $SECOND ";

if($zone eq 'UTC/UTC')
  { return( &PrettyDate(1,$YEAR,$MONTH,$DAY, $HOUR, $MINUTE, $SECOND) );
  };

  ## print "Debug: fetching tz date: '$zone'\n"; # 2021-03-13
if(length $zone < 1)
  { $zone = 'UTC';
  };

  my($yours)='';
eval
  { my($dt1) = DateTime->new( year => $YEAR, month => $MONTH, day => $DAY,
       hour => $HOUR, minute => $MINUTE, second => $SECOND,
       time_zone => $zone);
    my($dtest) = $dt1->clone->set_time_zone( 'UTC' );
    $yours = $dtest->datetime();  # format is identical
    1;   # prevents alterations to $@
  } or do
  { my($e) = $@;
    chomp($e); # $e should contain 'bad' zone name.
    my($dt2);
    my($woops);
   if($e =~ /The timezone .* could not be loaded/)  ## e.g. new zone like America/Coyhaique
     { $woops = "Missing timezone in tz DateTime: '$e' --- defaulting to UTC\n";
       $zone = 'UTC';
     } else # Walldate is for debugging:
     { my($walldate) = &PrettyDate(0,$YEAR, $MONTH, $DAY, $HOUR, $MINUTE, $SECOND);
       $woops = "BAD zone time, Perl DateTime said: '$e'; wall=$walldate";
       ($YEAR, $MONTH, $DAY, $HOUR, $MINUTE, $SECOND) = Timely::FullGregorian($JD, 0, 0, 1);
     };
     $dt2 = DateTime->new( year => $YEAR, month => $MONTH, day => $DAY,
           hour => $HOUR, minute => $MINUTE, second => $SECOND, time_zone => 'UTC');
     $yours = $dt2->datetime() . '?';
     my($corrected) = $dt2->clone->set_time_zone( $zone )->datetime();
     &Warn(3, "$woops; corrected='$corrected' " );
  };
```

```
##  print " => $yours } ";

  return($yours);
}
```

6.2.4 Internal Julian

Given a Gregorian date specific to a given zone, find the internal (GPS) rendition of the date. The returned value is a Julian number as *seconds*. Assumes that MYTIMELY has already been set up, allowing 'quick' G2J ().

```
sub InternalJulian #
{ my($inp, $region, $regionNAME) = @_;

  my($YY,$MM,$DD,$h,$m,$s);

if($inp =~ /^\s*(\d{4})-(\d{2})-(\d{2})[ T](\d{2}):(\d{2}):(\d{2})\s*$/ )
  { ($YY,$MM,$DD,$h,$m,$s) = ($1, $2, $3, $4, $5, $6);
  }
elsif( $inp =~ /^\s*(\d{4})-(\d{2})-(\d{2})\s*$/ )
  { ($YY,$MM,$DD, $h,$m,$s) = ($1, $2, $3, 0,0,0);
  } else
  { print " [ERROR] date format is YYYY-MM-DD hh:mm:ss \n";  # ? be more lenient
    return(0);
  };

  my($J, $shadow) = (0,0);
if($region == $UTCCODE)  # if UTC
  { $J = &GpsJulian($YY,$MM,$DD,$h,$m,$s, 0, 0, 0);
  } else
  { ($J, $shadow) = &G2J(1, $region, $YY,$MM,$DD,$h,$m,$s);
  };
if( ( $shadow < 0)   # failed, -ve shadow
  || ! $J            # or zero (bad J)
  )
  { print "\nBad date for zone $region($regionNAME): $inp ("
          . &JulianDay($J) . " / " . $shadow . " )\n";
    return(0); # fail
  };
```

Note that the external TZ date routine automatically jumps forward when it is supplied with a timestamp within the groundhog hour, on conversion to UTC. This places the discontinuity at the start of the groundhog time, while my program puts it at the end. In other words, TZ jumps forward by +DST at the start of the groundhog hour. This discrepancy results in a mismatch between my Greg () routine and FetchTzDate (). Try e.g. America/Phoenix 1943-12-31 23:01:00 versus 1 minute earlier.

```
  my($mine) = &Greg($J,0,0,1);  # Render as standard date, then do the same in TZ!
  $mine =~ s/ /T/;  # replace ' ' with 'T'
```

```
  my($yours) =  &FetchTzDate($J, $regionNAME, $YY, $MM, $DD, $h, $m, $s);

  my($yourmsg);
  my($err) = '';
if(length $yours < 1)
  { $yourmsg = "[no reference]";
  } else
  { $yourmsg = $yours . $TICK ;
  if($mine ne $yours)
    { my($mine2) = &Greg($J + $shadow, 0, 0, 1); # add s; compensate for TZ jump!
      $mine2 =~ s/ /T/;
    if($mine2 eq $yours)
      { $err = ' (+) ';
      } else
      { $err = " Date mismatch '$mine'??  vs ";
      };
    };
  };

if(length $err > 0)
  { print sprintf("%.12f", &JulianDay($J) )
          . " zone=$region (shadow=" . $shadow . " s) $err gps=$yourmsg" . "\n";
#  }
#elsif($FULLDEBUG)
#  { print sprintf("%.12f", &JulianDay($J) ) . " zone=$region (shadow="
#           . $shadow . " s) gps=$yourmsg" . "\n";
  };
  return($J);
}
```

In the above, FetchTzDate () retrieves a Gregorian date formatted by the DateTime Perl module (which uses tz); the value in `$J` and the associated routines are checked by applying Greg () back to the value in `$J` that was set by G2J () above. If there's an error, this might have been introduced in two places: either the Julian conversion by G2J () *or* the back-conversion by Greg ().

6.2.4.1 Julian Day

Convert seconds to Julian day number: a convenient wrapper.

```
sub JulianDay #
{ my($s) = @_;
  return($s/86400.0);
}
```

6.3 Full Gregorian

Given:

jd A Julian day value, expressed as seconds (rather than the more conventional 1=1 day, with fraction)

LCL a local offset, in seconds

DST a daylight saving time offset in seconds

isgps Whether this Julian day includes a GPS offset (i.e. it's a proleptic Julian GPS time).

... return a list of year, month, day, hours, minutes and seconds—for the corresponding Gregorian value, using the *supplied* local offset and daylight saving. We make no representation that the supplied values are right—we simply use them.

If $isgps is set to 1, then we strip off a GPS offset from the Julian day *before* we convert to Gregorian values.

We use Peter Baum's translation algorithm.[1] We currently just add DST and the local offset.

Equation	Baum step
Z = INT(JD - 1721118.5)	step 1
R = JD - 1721118.5 - Z	
G = Z - .25	used in later steps
A = INT(G / 36524.25)	step 2
B = A - INT(A / 4)	part of step 3
year = INT((B+G) / 365.25)	part of step 3 and step 4
C = B + Z - INT(365.25 * year)	step 5
month = FIX((5 * C + 456) / 153)	step 6
day = C - FIX((153 * month - 457) / 5) + R	step 7 and 8
IF month > 12 THEN	step 9
year = year + 1	
month = month - 12	
END IF	

In the following `$jd` is now in *seconds*, as are `$LCL` (Zone offset) and `$DST`.

```
sub FullGregorian #
{ my($jd, $LCL, $DST, $isgps);
    ($jd, $LCL, $DST, $isgps)=@_;

  # ensure $jd is in integer seconds, and likewise for LCL, DST:
  $jd = &Biggen($jd);
  $LCL = &Biggen($LCL);
  $DST = &Biggen($DST);

if($isgps)
  { $jd = &ExGps($jd);  # get rid of GPS offset, still integer.
  };

  my($Z, $R, $G, $A, $B, $C);
  my($year, $month, $day);
```

[1] https://www.researchgate.net/publication/316558298_Date_Algorithms

```
  $jd += $LCL+$DST;

  my($ss);
  my($hh, $mm);

    $Z = floor($jd/86400 - 1721118.5);       # Z = integer Julian days (exact)
    $R = $jd - 43200*(3442237 + 2*$Z);       # 1721118.5*2 = 3442237
      # R is the rest (as seconds, integer)

    # the following are calculations in terms of days, including leap years etc.:
    $G = $Z - 0.25;                          # 0.25 is precise, $G precise (days)
    $A = floor( ($G) / 36524.25);            # $A is integer
    $B = $A - floor( ($A) / 4);              # $B is integer
    $year = floor( ($B+$G) / (365.25) );     # $year is integer
    $C = $B + $Z - floor(365.25 * $year);  # $C is integer
    $month = int( (5 * $C + 456) / 153 );    #

    $day = $C - int( (153 * $month - 457) / 5 ) + $R/86400; # note $R use
  if ($month > 12)
     { $year = $year + 1;
       $month = $month - 12;
     };

    my($hms) = 24*$R;                 # 24*seconds !!
    $hh =int($hms/86400);             # hours
    my($ms) = 60*($hms - $hh*86400);  # minutes AND seconds
    $mm =int($ms/86400);              # minutes
    my($sx) = 60*($ms - $mm*86400);
    $ss = int($sx/86400);

    # a redundant check:
if($ss - $sx/86400 != 0) # should always be exact, check:
  {
    print "\n***BLAST!!*** $ss, $sx : delta="
          . sprintf("%.12f", ($ss - $sx/86400) );
  };

  return ($year, $month, int($day), $hh, $mm, $ss );
}
```

6.3.0.1 ExGPS

Given Julian day number as seconds, remove GPS adjustment (leap seconds). This assumes `@LEAPDATA` has been populated.

The value supplied in J is in seconds. `@LEAPDATA` contains (in order) a gps timestamp, a utc timestamp, and the offset in seconds. To convert from GPS to utc, subtract the offset value if the time is greater than or equal to the cutoff time; otherwise move to the next row and repeat, finally adding 9s if no match is found (The table has the most recent value as its *first*

entry).

```
sub ExGps #
{ my($J);
    ($J)=@_;

  my($Ltop) = scalar @LEAPDATA;
  my($i) = 0;
while($i < $Ltop)
  { my($d) = $LEAPDATA[$i];
    my(@D) = @$d; # dereference # [clumsy]
  if( $J >= ($D[0]/1000000.0) )        # [check this inequality ??]
    { return($J-$D[2]); # days
    };
    $i ++;
  };
  return($J+9);
}
```

6.3.1 Invoking FullGregorian ()

Because of the central position of this function and the first argument, let's look at how it's invoked. The callers are:

FetchTzDate () Invoked by:

> **InternalJulian ()** Implements g, u [end]
> **Multitest ()** implements v [end]

ListLeapseconds () *Trivial*

J2G () — Invokers:

> **MultiTest ()** implements v, as above [end]

Greg () Multiple uses:

> **ReadLeapseconds ()** A simple check. [end]
> **InternalJulian ()** Implements g, u [end], as above.
> **Multitest ()** implements v, as above [end]

QuickJ2G () *?? — of dubious current utility, needs work.*

6.3.2 Fix date fraction

Remove fractional seconds from date; SQL server chokes if more than 3 digits. [? utility]

```
sub ClipFractionalSeconds #
{ my($d) = @_;
if($d =~ /(.+)\.(\d+)/ )
  { $d = $1;  # clip everything after the decimal, AND the decimal point
  };
  return($d);
}
```

6.3.3 Single Gregorian

The value in $J is Julian, but in seconds; `$LCL` and `$DST` likewise.

```
sub Greg #
{ my($J, $LCL, $DST, $isgps) = @_;

if($J < 1) # rubbish
  { return('_');
  };
  my($YY,$MM,$DD, $h,$m,$s) = &FullGregorian($J, $LCL, $DST, $isgps);
if($YY < 1) # 1 AD
  { return('?'); # we're not interested in BC!
  };
  return( &PrettyDate(0,$YY,$MM,$DD,$h,$m,$s) );
}
```

6.3.4 Pretty date

If the initial value is non-zero (i.e. 1) use 'T' as the separator between date and time, rather than ' '.

```
sub PrettyDate #
{ my($isT, $YYYY,$M,$D,$h,$m,$s) = @_;
  my($spacer) = ' '; # usual separator between date and time
if($isT)
  { $spacer = 'T';
  };
  return( "$YYYY-" . &DoubleDigit($M) . '-' . &DoubleDigit($D) . $spacer
    . &DoubleDigit($h) . ':' . &DoubleDigit($m) . ':' . &DoubleDigit($s) );
}
```

6.3.4.1 Microseconds to Julian

The converse, now simply converting microseconds to seconds, rather than a Julian day number or part thereof.

```
sub HugeToJ #
{ my($d);
    ($d)=@_;
if(! defined $d)
  { return(PART_EMPTY);
  };
if($d !~ /^-?\d+/ )  # allow -ves
  { return('_')
  };
  ## print "HugeToJ took $d gave " . sprintf("%.12f", $d/1000000) . "\n";
  return( &Biggen($d/1000000) ); # convert microseconds to *int* seconds [NB]
}
```

6.4 Julian to Gregorian

Given J, a GPS timestamp, convert this to a Gregorian date pertinent to that location (wall clock time). As all of the stored values in the relevant row of the database use GPS time, comparisons are relatively simple, and we don't adjust for leap seconds until just before we convert to Gregorian.

1. Find the location code and year. Get the relevant line for this location and year from **daytime**. Note that if we are within a few hours of the start or end of the year, we may have a problem, as we will only be certain that we are within the "wall clock year" after we've gone through the whole of the following calculation! So we may determine that we've chosen the wrong year *after* we've done all of the calculations, and may have to repeat with the following or previous year! [EXPLORE THIS ANOMALY, might facilitate by having a third redundant timestamp that signals the start of the year in wall time, and even a fourth "year end"]
2. Record the **zone_offset** as Z, and if **zone_transition** is nonzero:
   (a) If J >= **zone_transition**, set Z to **zone_future**
3. Create D=0, the daylight saving value. Now, if **dst_start** is non-zero:
   (a) If **dst_end** is zero:
      i. if J >= **dst_start**, D = **dst_add**;
   (b) else, if **dst_end** > **dst_start**, we are in northern hemisphere:
      i. if J >= **dst_start** and J < **dst_end**, D = **dst_add**;
   (c) else, must be in southern hemisphere:
      i. if J < **dst_end**, D = **prior_add**;
      ii. if J >= **dst_start** D = **dst_add**;
4. Else (**dst_start** is zero) if **dst_end** is non-zero:
   (a) if J < **dst_end**, D = **prior_add** (Carried over from previous year)
5. Adjust J for leap seconds using the relevant GPS time cutoff; to J add Z the relevant **zone_offset** and D dst value (add) and make the actual timestamp.

6.4.1 J2G

The submitted `$jd` value is GPS time in seconds. We adjust for zone and DST for this time and `$zone`. By submitting `$qic=1` to Anomalous (), we can speed things up a bit.

Returns a plethora of values:

$YY, $MM, $DD, $h, $m, $s Obvious from context

$LCL Z value in seconds, currently obtaining for this date

$DST DST value in seconds, likewise.

$hog Whether we're in the zone ("groundhog zone") where Julian values are ambiguous, because we've fallen back, either due to cancellation of DST, or a zone change. Is -1 if the Julian value is in the shadow zone; is +1 if we're in the "fall-back" area.

$delta_Z The amount of zone change in seconds

$delta_DST The amount of DST change, in seconds.

```
sub J2G #
{ my($qic, $zone, $jd) = @_;

  my($J) = $jd * 1000000;  # [really should use big number libraries]


  my($YEAR, $xMM, $xDD, $xh, $xm, $xs) = &FullGregorian($jd, 0, 0, 1);  # NB $jd NOT $J
    # clumsy, rather just divide etc. [fix me]
  # print "Debug: J2G : $zone $jd\n";


  my($Z,$D, $hog, $deltz, $delds) = &Anomalous($qic, $zone, $YEAR, $J);
  # NB. $deltz, $delds are in microseconds, at present.


  my($LCL) = $Z/1000000; # microseconds to days
  my($DST) = $D/1000000; #
  my($YY, $MM, $DD, $h, $m, $s) = &FullGregorian($jd, $LCL, $DST, 1);  # GPS *on*

# print "Debug: J2G done: $YY, $MM, $DD, $h, $m, $s\n";

  # return all values, not a composite:
  return( $YY, $MM, $DD,  $h, $m, $s,  $LCL, $DST, $hog, $deltz/1000000, $delds/1000000);
}
```

6.4.2 Handle anomalies

Recall that in timely, all transitions refer to the past, and associated Z and DST values apply until the transition. Anomalous () retrieves relevant entries for zone and year. If nil found, fail; otherwise iterate through until transition value above Julian. We know the DST and zone values then apply. Anomalous () is only invoked by J2G (). The arguments are:

$handDB Database handle

$zone Internal zone code

$YEAR eponymous

$JBIG The Julian timestamp we wish to characterise: this is a big integer in microseconds after -4712-1-1 12:00:00 i.e. January 1st, 4713 BCE.

The return values are several:

$z The *current* Z for this timestamp;

$d The *current* DST for this timestamp;

"$hog" Whether the timestamp is within the "shadow" (groundhog) zone where two Julian values map to one wall time. This code has internal structure:

0 Not in shadow

-1 In shadow, the first of the two Julian values that map to a single wall timestamp

1 The second shadow

$deltaz if $hog is non-zero, then the pertinent change in zone offset that generated the relevant shadow; otherwise zero

$ddst likewise.

For dealing with a $JBIG value that's close to a transition T, it's important to identify whether the value is within the "groundhog" period after the transition from a higher DST value D[-1] to a lower one D[0], in other words if:

- D[0] < D[-1] **and**
- T < JBIG < T+D

This is however a bit more nuanced, because logically the antecedent period is similarly ambivalent! A single "wall time" value maps to *two* UTC (or GPS) timestamps. There's yet another complication: if there's a similar transition from a given zone offset to a smaller zone offset, at the transition we'll jump back in time by the difference, with another "groundhog hour". We also need to accommodate this.

Another issue is where we're at the (early) cusp of a transition, for example at the start of the shadow hour when DST of +1 hour is about to be turned off. The -1 hour mark must be included in this shadow hour—otherwise we'll get an error for every v-3600 (or whatever).

We also return the actual delta values for zone and time, where appropriate

Now consider the following example (Europe/Minsk in 1941): at a wall time of 1941-06-28 00:00:00, corresponding to UTC 1941-06-27 21:00:00, we drop z from 3 to 1 hours and add DST of 1 hour. Effectively we jump back by 1 hour to a wall time of 1941-06-27 23:00:00 (UTC 1941-06-27 20:00:00 also corresponds); the shadow period is thus calculated by deltaDST-deltaZ = 1 -2 = -1. any wall time between 23:00 and 00:00 of the next day is in the shadow, corresponding to UTC 1941-06-27 20:00:00 to 22:00:00.

```
sub Anomalous #
{ my($qic, $zone, $YEAR, $JBIG);
    ($qic, $zone, $YEAR, $JBIG)=@_;

if($zone == $UTCCODE)
```

```
  { return(0, 0, 0, 0, 0);  # Z=0, DST=0, not groundhog, ...
  };  # [explore: what it $zone is zero??]

if( $qic
  &&($zone != $MYZONE) # check it's the right region, D'Oh!
  )
  { &Aagh("Quick Anomaly, mismatched zones: $zone|$MYZONE", __LINE__, 0);
    return;
  };
```

Prepare SQL, as needed:

```
  ## my($epsilon) = 500000; # microseconds 0.5 seconds [? : see notes]
  my($epsilon) = 10000; # 10 microseconds

  my(@ROWS);
  my($hi) = 2 + int($YEAR);
  my($lo) =-1 + int($YEAR);
if($qic)
  { (@ROWS) = &QuickBetween($lo, $hi); # as for G2J_(): get tr, dst, zo, ignored
  } else
  { my($handDB) = $DB_MAIN;
    my($q) = "SELECT transition, dst, zone_offset, ignored from timely "
                                    # a dummy value ^
    . "WHERE region = $zone AND year BETWEEN $lo AND $hi ORDER BY transition";  # ASC
    (@ROWS) = &SQLManySQL($handDB, $q, 'get all rows');
  };
```

In the above SQL, because there ia always a transition at year end, we are guaranteed values in the preceding and following years, to act as "stoppers", unless the submitted date is "out of range"—too low or too high for there to be stoppers. To accommodate year-end transitions, the `$hi` value adds not one but two years.

We now find where $JBIG lies. In the **timely** table, the Z and DST values apply prior to any transition. Because two Julian values may map to a single wall time, there may be gaps in the wall time, but there should never be an unmapped Julian value—this would imply a Julian outside the range of the database. We will not only return the Z and DST values that apply to the submitted Julian, but also signal (in "$hog", the third value returned) whether there is a second Julian value that would also have given the same wall time!

We:

1. Work through all of the rows, until $JBIG is below the transition time. Initially, because the transition times are sorted in ascending order and we work upwards, $JBIG will be above the current transition (If we've input a very low value of $JBIG, then the first row will apply, willy-nilly). If we reach the end without the condition being met, then we have a problem: our date is out of range.

   (a) Another way of saying $JBIG is below is that $tr - $JBIG > 0. Actually, I use a slight value over 0, $epsilon, e.g. half a second or

less. This means that if $JBIG is exactly at the interface, the next transition applies, rather than the current one.

2. We now know that the row (with its Z, DST) applies to our time.

A few initialisations, and we're in a large `while` loop that performs the test just described:

```
  my($r);
  my($trold) = 0; # preceding transition
  my($dold) = 0;  # preceding DST value
  my($zold) = 0;  # preceding zone offset

  my($i) = 0;
  my($rL) = scalar @ROWS;
while($i < $rL)
  { $r = $ROWS[$i];
    my($tr, $d, $z, $dmy) = @$r;  # database tr (Julian), DST, Z that apply up to $tr.

  if( ($tr - $JBIG) > $epsilon )  # we have our transition row: A large IF CLAUSE
    { ## print "\n[$JBIG : $tr], diff=" . ($tr - $JBIG) ;
```

Now the value in JBIG is somewhere in between the previous transition ($trold) and the current transition ($tr):

```
    previous transition > || xxx | ... | yyy || < current transition.
```

The DST and Zone parameters $d and $z relate to this interval; the values in $dold and $zold relate to the previous one. There are two rather special cases:

1. JBIG is within the section indicated by xxx—and xxx reflects a "groundhog hour" where this Julian value and another value both map to the same wall time (this is the second Julian, and will be signalled by $hog=1), around the *previous* transition. The change in Z and/or change in DST that brought this about relate to the difference between the current values and the preceding ones.

2. JBIG is within the section indicated by yyy, another groundhog time, but this time, the first Julian that maps to the groundhog hour about the current transition. This will be signalled by $hog = -1. The change in Z and/or DST relate to the current transition, and as such, we need to examine *future* Z and DST values from the subsequent transition!

First, the simpler case of $hog = 1. As a convenience, we identify section xxx by adding the change in Z and DST (let's call this $shadow):

```
    my($deltaz) = $z - $zold;   # eg. will be negative if Z has decreased
    my($ddst) = $d - $dold;     # eg. will be positive if spring forward
    my($shadow) = $deltaz+$ddst;
    ## print " shadow=$shadow ($deltaz $ddst [$z $zold $d $dold]) ";
```

If the shadow is negative, then there was a jump back at and subsequent to the previous transition. To determine whether JBIG falls into section xxx, we thus *subtract* $shadow from $trold, and only respond if JBIG is less than this.

```
if($JBIG < $trold-$shadow) # we know $JBIG is > $trold.
  { ## print "Debug Anomalous_() z=$z d=$d\n" ;
    return($z, $d, 1, $deltaz, $ddst);
                # ^ "$hog"
  };
```

The second case, where JBIG is in section yyy, is a bit more tricky. We need to fetch the *following* values to calculate deltaz and ddst!

```
    $i ++;
  if($i >= $rL) # run out!!
    { &Warn(3, "Anomalous_() data ran out: zone=$zone, year=$YEAR, value=$JBIG");
      return($z, $d, 0, 0, 0); # or might even fail
    };

    $r = $ROWS[$i];  # next row
    my($tnext, $dnext, $znext) = @$r;

    # a further caution: ? can be 'next' transition almost immediately after
    # current--- related to our end-of-year 'ignorable' insert.
    # We thus check for this:
  if( ($tnext - $tr) < 120000000) # currently use 2 minutes, NB Phoenix
    { $i ++;
    if( $i >= $rL )  # [can this happen?]
      { &Warn(3, "Upper adjustment failed!! zone=$zone, year=$YEAR, value=$JBIG");
      } else
      { print '>>'; # [hmm]
        $r = $ROWS[$i];  # next row
        ($tnext, $dnext, $znext) = @$r; # throw away the rubbish
      };
    }
    # end check.

    my($nextdz) = $znext - $z;
    my($nextdd) = $dnext - $d;
    my($nextshadow) = $nextdz + $nextdd - $epsilon;
    # epsilon extends the shadow slightly back to cover the e.g. -1hr mark.
    ## print " next shadow=$nextshadow ($nextdz $nextdd [$z $znext $d $dnext]) ";
  if($JBIG > $tr+$nextshadow) # will only work if $nextshadow is -ve
    { ## print "Debug Anomalous_() z=$z d=$d\n" ;
      return($z, $d, -1, $nextdz, $nextdd);
                  # ^ "$hog"
    };
```

Otherwise, we've found our position, and values for Z and DST, and can simply return these. There is no groundhog issue.

```
        ## print "Debug Anomalous_() z=$z d=$d\n" ;
        return($z, $d, 0, 0, 0);
            # both zero ^  ^ if hog is zero [explore]
    }; ## END large IF CLAUSE.
```

The end of the large `while` loop:

```
    $trold = $tr;
    $dold = $d;
    $zold = $z;
    $i ++;
  }; # end while.

  # [EXPLORE THIS FAILURE]
  &Warn(1, "Timestamp " . $JBIG/86400000000 . " out of range zone=$zone, y=$YEAR");
  return(0, 0, 0, 0, 0);
}
```

Exiting here is disappointing: our translation has effectively failed.

6.5 End off

In conclusion, I've learnt a few things:

1. Timezones are crazily complex and challenging to implement;
2. I was initially way off, in simply "going with the Perl flow" and with convention. I initially used floating point Julian day values. Far smarter is to rely on integer seconds, the problem being with Perl's failure to distinguish between integers and floating point.
   (a) Where a small floating point different results from division, it is wise to immediately coerce this back to an appropriate integer, being aware of the danger of truncation.
3. A corollary of (2) that isn't immediately obvious is this:
   Fractions of a second are the same in Julian and Gregorian space. So to convert between them:
   (a) First, strip off the partial second;
   (b) Then do the integer conversion to seconds in the relevant space (with appropriate checks and attachments);
   (c) Finally add the partial seconds back again!

```
################################################
#              END OF MAIN ROUTINE             #
################################################
1;
```

The `1;` *at the end of the module* is important as it prevents eval from failing in Perl.

7.0 Setup

This has been retained, but also read the start of *small_time_400.lyx*. The following is all done there, so rather follow those instructions, and regard all of this chapter as an FYI.

1. Make the following subdirectories in your current directory (e.g. within */w/seekwell/* or *~/smalltime/*)
   (a) seek
   (b) perl
   (c) perl/lib
   (d) perl/log
   (e) perl/tz
   (f) perl/csv
2. Install a LAMP, WAMP or MAMP stack, and then install ***f*ehr**. Although Timely depends on just a few ***f*ehr** tables (**leapseconds**, **timely**, **PLACES**, **countrycodes**, **SKEYS** and the tables they in turn depend on—**PEOPLE**, **SOURCES**, and thence **Sourcescripts** and **Institutions**) it's smart to install the the whole schema.
3. From the command line, open MySQL (mysql -u root -p followed by enter and password) and use fehr;
   (a) Ensure that *empty* **leapseconds** and **timely** tables are present. They look like this:

```
CREATE TABLE leapseconds
( leapsecond int
    ,constraint leapkey PRIMARY KEY (leapsecond)
  ,gpstime BIGINT
  ,utctime BIGINT
  ,toffset int -- baseline is -9
  ,ver integer default 0
  ,chk int
)CHARACTER SET=utf8mb4
    COLLATE=utf8mb4_unicode_ci;SHOW WARNINGS;DESCRIBE leapseconds;
```

   and:

```
CREATE TABLE timely
( timekey integer
    , constraint timely_pk PRIMARY KEY(timekey)
  ,region BIGINT
    ,constraint timely_region_fk FOREIGN KEY (region)
      references PLACES(place)
  , year integer
  , transition BIGINT
  , zone_offset BIGINT
  , dst BIGINT
  , ignored integer default 0
  , ver integer default 0
  , chk int
)CHARACTER SET=utf8mb4
    COLLATE=utf8mb4_unicode_ci;SHOW WARNINGS;DESCRIBE timely;
```

Other tables that will be used by Timely are mentioned above. For creation code, see *fehr_sql_400.lyx*.

4. Update the countrycodes table to include a 'UT' code (for universal time), if not already present:

```
INSERT INTO countrycodes (country, ccode, tla) VALUES (1, 'UT', 'UTC');
```

5. ***Create*** the following ancillary rule tables in ***f*ehr**. There is minimal processing, and the tables are denormalized with respect to rule_name.

   (a) rule sets:

```
CREATE TABLE tz_rules
( tz_rule integer
  ,rule_name varchar(16)
  ,from_year varchar(8)
  ,to_year varchar(8)
  ,in_on_at varchar(64)
  ,dst varchar(8)
  ,is_active integer default 1
)CHARACTER SET=utf8mb4
    COLLATE=utf8mb4_unicode_ci;SHOW WARNINGS;DESCRIBE tz_rules;
```

   (b) zone cutoff rules

```
CREATE TABLE tz_cutoffs
( tz_cutoff integer
  ,stdoff varchar(16)
  ,rule_name varchar(16)
  ,region BIGINT
    ,constraint tz_cutoffs_region FOREIGN KEY (region)
      references PLACES(place)
  ,until_text varchar(16)
  ,is_active integer default 1
```

```
)CHARACTER SET=utf8mb4
    COLLATE=utf8mb4_unicode_ci;SHOW WARNINGS;DESCRIBE tz_cutoffs;
```

(c) To prevent duplicates and check, each rule row can be identified not just by the sequential, internal PK in tz_rule or tz_cutoff, but respectively by rule_name+from_year+to_year+in_on_at; and region+until_text. This allows us (at the start of any re-importation) to:

i. Reset all active values to 0
ii. Re-activate all identified rows to 1
iii. Add new rows, as needed, in either table.

(d) We also need source rows in **SKEYS**:

```
INSERT INTO SKEYS (kName,        kValue, klock) VALUES
                  ('tz_rules',   1000,   0),
                  ('tz_cutoffs', 1000,   0);
```

6. Download the tz database (as a zip, from https://www.iana.org/time-zones) and unzip it in ...*seekwell/perl/tz* ; the name will be along the lines of *tzdata2020d.tar.gz*

   (a) Unzip the contained files (if under Windows, using an application like 7zip) and put them in *perl/tz* so that files like *africa* and *LICENCE* are immediately within this directory.

7. Turn the current file (*timely_400.lyx*) into a Perl package:

   (a) First export *seek_400.lyx* to *seek_400.tex* and *timely_400.lyx* to *timely_400.tex*.

8. Run Dogwagger over these TeX files. You can obtain Dogwagger from https://github.com/jvanschalkwyk/dogwagger. In the containing directory say:

```
perl Dogwagger405.pl timely_400.tex
perl Dogwagger405.pl seek_400.tex
```

   (a) Once you've done this, the *make.bat* file should be produced, and henceforth under Windows you should be able to just say `make` ; similarly use `bash make.sh` under Linux.

9. Under Linux, you will need to install

   (a) Term::Readkey

```
sudo apt-get update
sudo apt-get install libterm-readkey-perl
```

   (b) DBI — this may be slightly traumatic (See https://dr-jo.medium.com/the-fourth-circle-a2831bbdf769).

```
sudo apt-get install libdbi-perl
sudo apt-get install libdbd-odbc-perl
```

   To check your sources, run: `odbcinst -j` , which will point you to eg

/etc/odbc.ini, as well as
/etc/ODBCDataSources and
/home/<mynamehere>/.odbc.ini i.e. ~/.odbc.ini

`printenv ODBCSYSINI` should show you this environment variable, which should point to *odbcinst.ini* ; similarly ODBCINI should refer to *odbc.ini* (or printenv | grep ODBCSYSINI)

```
export ODBCSYSINI='/home/jo'
export ODBCINSTINI='.odbcinst.ini'
export ODBCINI='/home/jo/.odbc.ini'
```

odbcinst -q -d

(c) DateTime

```
sudo apt-get install libdatetime-perl
```

10. You can obtain the DateTime version (which is important) by saying:

```
perl -M"DateTime 9999"
```

This should be compared with the most recent version available (that should correspond to the most recent version of tz available from IANA (https://www.iana.org/time-zones). Note however that this will be something like 2021a; the corresponding DateTime version should be on CPAN (https://metacpan.org/pod/DateTime) but this may lag. In the above example, the date for v 1.54 is 2020-12-04, so this won't contain the first 2021 tz update. As your perl DateTime version diverges, the error count in timely will rise.

11. Move to e.g. */fehr/seekwell/perl* and run

```
perl seek.pl
```

7.1 "Design features"

Some readers will be disturbed by:

- My tendency not to use standard 'code-shop' indentation of braces;
- Globals like `$DB_MAIN`. You might argue that this should be tossed around as a parameter rather than the approach taken—to instantiate `$handDB` locally from the global, as needed close to use of SQL primitives that applies just to this database.

This is the only apology you'll get—and it's not much of an apology. These are deliberate; all of the other design flaws are made from sheer ignorance.

7.2 Residual issues

1. Perl automagically converts integers over about 2^{54} to floats (actually, they're all floats, it just doesn't show well), and this results in the best case in loss of precision. If the number is a primary key,

effectively all is lost. Solutions exist, but bignum (bigint + bigfloat) has a number of side effects, including slower processing. If you try simply to use bigint, then all floats are coerced to integers, which may not be, shall we say, highly desirable. PK big integers should thus be represented as strings, and kept as strings.

2. It will be wise to timestamp t_start and t_amended within the two tz_* tables in ***f*ehr**. I haven't yet done this.

3. Small things, cf:

   (a) "# no GPS adjust so $DST is a redundant variable [fix me]"

7.3 Change log v2

1. Version 2.0.1 (2,000,001):

   (a) It's important to document the requirement for a UT entry in countrycodes. See point 4 at the start of Chapter 7, page 68.

   (b) It's similarly important (Page 70) to document the current DateTime version and if possible update this. Doing this under Windows, where ActivePerl at least lags conspicuously, may not be easy.

   (c) On testing under Windows 7, Z and DST values were initially reported with decimal fractions [explore]

   (d) IT WILL BE WISE to have creation/update times not only in tz_ files but even in timely [explore]

   (e) Have note on how to upgrade DateTime where you can (mainly, Linux).

      i. Also note crash: **Execution trapped (debugging)The timezone 'America/Nuuk' could not be loaded, or is an invalid name.

2. On 2021-03-07 made CheckOS () public.

3. On 2021-03-14 added soft option to TooBig (3.5).

4. On 2022-02-07 modified XPrint () to prevent attempt to write to log once handle is closed.

7.4 Change log v4

Enormous: moved out huge parts. See *small_time_400.lyx*.