# small time

## Launcher for ‘timely’

Version 4,000,000

J. M. van Schalkwyk

31st January 2026

# Contents

# 1.0 Introduction

I built the **timely** Perl module for use within my data extractor scripting language **vector seekwell**, but it was convenient (especially for pedagogic purposes) to have a simpler stand alone that makes it easy to launch the Perl code that does the extraction of rules from IANA's tz, and the corresponding translation into an SQL table.

I then realised that it was more sensible to shrink the library module, and move the one-off or seldom-used stuff to this Perl program. This required a fair bit of refactoring, and could still be done a lot better. (Volunteers?)

## 1.1 Basic needs

You will need:

1. Perl (e.g. Strawberry Perl, if you're on Windows; Perl is native to Linux.)
2. MariaDB or MySQL installed (under Windows, use WampServer) as well as an ODBC connection, properly set up. For details, see Section 3.1.
   (a) It's wise to go to http://localhost/phpmyadmin/ and log in as `root`, noting that by default there's no password! Set a secure password under User accounts | Edit privileges (for root) and [Change password].
   (b) You may wish to create a subsidiary 'vanilla' user on localhost with limited privileges i.e. SELECT, INSERT, UPDATE to use for your connection.
3. A recent copy of the tz files from IANA https://www.iana.org/time-zones e.g. *tzdata2025c.tar.gz* with the ability to unzip it (using gzip or 7z)
4. My Perl script *Dogwagger405.pl* from https://github.com/jvanschalkwyk/dogwagger/
5. A directory into which you can place the .lyx file (e.g. *~/smalltime/* ) as well as the subdirectories
   (a) *smalltime/perl/* and
   (b) *smalltime/sql/* and
   (c) *smalltime/perl/lib/* (in this example) and

   (d) *smalltime/perl/tz/* (for the decompressed tz files) and
   (e) *smalltime/perl/log/*

6. A cut-down version of the ***f*ehr** database with just the required tables. This will be called **smalltime**. We'll build it below.
7. A copy of the timely.pm Perl module in the *lib* sub-subdirectory. The source code is in *timely_400.lyx*
8. The Perl script small.pl from Chapter 4. This simply:
   (a) Connects via ODBC to MySQL
   (b) Makes the appropriate calls to the *timely.pm* Perl module.

Let's start with that database.

# 2.0 Minimal database

You will first need to install MySQL, setting up a root password and so on. This chapter describes the SQL code you'll use to make the **smalltime** database. To extract the various files contained in this *literate* document, you'll need to export the source LyX file as .tex, and then run my Dogwagger script, obtainable from GitHub: https://github.com/jvanschalkwyk/dogwagger. Details are in Chapter 6.

## 2.1 Create database & tables

From the MySQL command line:

1. Create the database;

```
create schema smalltime CHARACTER SET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

2. Run the SQL script, specifying the correct source e.g:

```
use smalltime;
source /smalltime/sql/smalltime.sql;
```

This is a Windows example, but on Linux or Mac, the only change you'll need is to specify a different path to the file *smalltime.sql*. In Linux, use the path *~/smalltime/* (etc).

First we will need to generate the SQL in this file, extracting it as described in Chapter 6. The rest of this chapter is devoted to creating the target file *sql/smalltime.sql*, as follows.

```
-- The main SQL file is smalltime.sql

 SET default_storage_engine = INNODB; -- trust nobody

 SET @GLOBALS_UNLIMITED = 999999999999999999;
```

## 2.2 Key sources

The PHP Fetchkey () routine will be used to provide sequential keys. This routine requires an **SKEYS** table, from which the sequential keys are retrieved for each table:

```
SELECT Now() `Starting`,"SKEYS" `Table`;

CREATE TABLE SKEYS (
  kName varchar(32)
    ,constraint bad_key_pk primary key(kName)
  ,kValue BIGINT
  ,klock BIGINT
  ,kuser integer
  )CHARACTER SET=utf8mb4
    COLLATE=utf8mb4_unicode_ci;
```

This is a cut-down version of the ***f*ehr** list:

```
SET @MIN_NORMAL_KEY = 2000;

INSERT INTO SKEYS (kName, kValue, klock) VALUES
  ('SOURCES',            @MIN_NORMAL_KEY+100, 0),
  ('PLACES',             @MIN_NORMAL_KEY+1000, 0),
  ('PEOPLE',             @MIN_NORMAL_KEY+40000, 0),

  ('Institutions',       @MIN_NORMAL_KEY+2000, 0),
  ('Sourcecripts',       @MIN_NORMAL_KEY+3000, 0)
  ;
```

Codes between 100 and 999 should be reserved for test cases. Codes from 100–999 should not be used for real entries (But countries in the **PLACES** table are different).

### 2.2.1 Supplement

```
SET @MIN_TEST_KEY = 100;
SET @MIN_NORMAL_KEY = 2000;

INSERT INTO SKEYS (kName, kValue, klock) VALUES
  ('timely', @MIN_TEST_KEY-1, 0),
  ('relationships', @MIN_NORMAL_KEY, 0)
  ;
```

### 2.2.2 Access & timing

We need to be concerned about how the key tables will perform in providing multiple sequential keys. Potentially a requesting process might be locked out as multiple competing processes queue for sequential keys. There is also the problem where a process dies while a generator key is locked. These problems justify the lock field in the **SKEYS** table. Through careful management of this field, we can deal with failure, regardless of the database used (Another strong argument for *not* using auto-incrementing keys).

## 2.3 Crude timestamp conversion

These are Julian Day i.e. JD (), the latter returning a Julian time[1] in microseconds, based on proleptic GPS time; and Gregorian Time i.e. GT (), which converts the Julian day value just described to a Gregorian timestamp. The limitation here is that FROM_UNIXTIME is constrained. We will subsequently explore more sophisticated conversions. Specifically, see Section 5.16.

```
CREATE FUNCTION GT (b BIGINT)
  RETURNS CHAR(26) DETERMINISTIC
  RETURN FROM_UNIXTIME( b/1000000 - 210866760000 );

CREATE FUNCTION JD (s VARCHAR(26))
RETURNS BIGINT DETERMINISTIC
RETURN 1000000*(86400*(1721059+TO_DAYS(s))+TIME_TO_SEC(s));
```

## 2.4 The REASONS table

We need a reason for any alteration. Here's the table:

```
SET @SYSOP = 2000;

SELECT Now() `Starting`,"REASONS" `Table`;

CREATE TABLE REASONS
( rsn integer
    ,constraint reason_pk PRIMARY KEY(rsn)
  ,description varchar(128)
  ,shortname varchar(32)
  ,t_amended BIGINT
  ,reason int
  ,p_amended BIGINT
  ,amender integer
--  ,constraint bad_reason_amender foreign key(amender)
--     references PEOPLE(person)
  ,srcID BIGINT
```

[1] Which we may refer to in the subsequent text as 'Julian day number', understanding that this includes a fractional day, and is expressed in microseconds.

```
  ,src BIGINT
--    ,constraint bad_reason_src foreign key(src)
--       references SOURCES(src)
  ,ver integer default 0
  ,chk int
) CHARACTER SET=utf8mb4
  COLLATE=utf8mb4_unicode_ci;SHOW WARNINGS;DESCRIBE REASONS;
```

The constraints on amender and src aren't yet created, as the relevant tables are still to arrive below.

The reason why we call the PK "rsn" is that we have a reason field (as in every other table) and even though this is a "meta-reason", we wish to preserve this structure.

We have the option of a 'null' reason entry (rsn=0), for imported data where no reason can be determined. This is also used where the reason is simply creation of the first entry; the choice of the description 'unknown' is perhaps less than well-inspired, but is shorter than 'not stated'. The following reasons have a pharmaceutical flavour, but more will be needed for other purposes.

```
INSERT INTO REASONS (
  rsn, description, t_amended, reason,
  p_amended, amender, src, chk, ver)
VALUES
  (0,  'unknown',   0, 0, 0, @SYSOP, 2001, 0, 0),
  (-1, 'deleted',   0, 0, 0, @SYSOP, 2001, 0, 0),
  ( 1, 'omitted',   0, 0, 0, @SYSOP, 2001, 0, 0),
  ( 2, 'given',     0, 0, 0, @SYSOP, 2001, 0, 0),
  ( 3, 'deferred',  0, 0, 0, @SYSOP, 2001, 0, 0),
  (10, 'abandoned', 0, 0, 0, @SYSOP, 2001, 0, 0)
;
```

The 'deleted' value with a primary key of -1 is very special, as it is used (carefully) to signal that a row is in error and has been deleted. This has substantial consequences for the handling of any row so flagged. Any negative value should be considered "erroneous".

## 2.5 The SOURCES table

Every line of every table will refer to a data source. This should permit clear identification of where data originated, and a check on its validity. (Every row also has the option of a chk value, but this is of limited value, and will not of course provide any security unless you replace it with a cryptographic hash). In the following we establish a convention used throughout the remaining code: show warnings and describe the table on creation. This facilitates debugging of the console-level MySQL output.

### 2.5.1 Source Scripts

We make the provision for also referencing a particular source script that was used to extract data. This is currently a facility that I've not used

much.

```
SELECT Now() `Sourcescripts`,"" `Table`;

CREATE TABLE Sourcescripts
( sourcescript BIGINT
  ,constraint bad_sourcescript_pk PRIMARY KEY(sourcescript)
  ,description varchar(64)
  ,version integer
  ,amender integer
--  ,constraint bad_sourcescript_amender foreign key(amender)
--    references PEOPLE(person)
  ,t_amended BIGINT
  ,reason int
  ,p_amended BIGINT

  ,src BIGINT
  ,srcID BIGINT
  ,ver integer default 0 -- version of this row
  ,chk int
)CHARACTER SET=utf8mb4
    COLLATE=utf8mb4_unicode_ci;SHOW WARNINGS;DESCRIBE Sourcescripts;
```

The `bad_sourcescript_amender` constraint should only be added once **PEOPLE** has been defined in Section 2.6. The script version should comprise $10^6$ * major version + $10^3$* minor version + subsidiary version number (ignored if not present). Do not confuse **version** with **ver**, the version of this row (that describes a script), a row that might conceivably have contained erroneous information and have been fixed, without any change to the actual script itself.

```
-- SET @SYSOP = 2000;

INSERT INTO Sourcescripts(sourcescript, description, version, amender,
   src, chk)
 VALUES(0, 'none', 0, @SYSOP, 2001, 0);
```

### 2.5.2 The actual SOURCES

Even the **SOURCES** table should have a src, but clearly the constraint needs to be added later![2]

```
SELECT Now() `Starting`,"SOURCES" `Table`;

CREATE TABLE SOURCES
( src BIGINT
  ,constraint bad_src_pk PRIMARY KEY(src)
  ,master varchar(128)
```

[2]Because it's possible that a source may describe more than one patient, or something other than a patient datum, it is not reasonable to "denormalise" as elsewhere by having a tag_person field.

```
  ,nature integer
  ,institution integer
--  ,constraint bad_source_institution foreign key(institution)
--     references Institutions(institution)
  ,script BIGINT
  ,constraint bad_source_script foreign key(script)
     references Sourcescripts(sourcescript)
  ,amender int
--  ,constraint bad_source_amender foreign key(amender)
--     references PEOPLE(person)
  ,t_amended BIGINT
  ,reason int
  ,p_amended BIGINT

  ,ssource BIGINT  -- check need for this field [explore]
  ,chk int

  ,ver integer default 0  -- version
)CHARACTER SET=utf8mb4
    COLLATE=utf8mb4_unicode_ci;
```

Note that source is a reserved keyword in SQL. The SOURCES table comprises several identities:

**master** a reference to a string that identifies the external 'master document'. This might for example be the name of a table in a database, or a document identifier.

**script** an identifier (including version) of the script used to parse the master;

**nature** describes the type of source. No associated table at present, but 0=undefined, 1=database

**institution** a foreign key that references the institution from which the document originates.

The `bad_source_institution` constraint cannot be added until the Institutions table has been defined. The `ver` field should rarely be used, if ever, although it's conceivable that a master value might be incorrectly specified and need amendment, a benign use of `ver`. We have a self-reference and a "0" value:

```
ALTER TABLE SOURCES ADD CONSTRAINT src_reflex foreign key(ssource)
  references SOURCES(src);



INSERT INTO SOURCES
 (src, master, institution, script, amender,
  t_amended, reason, p_amended, ssource, chk)
VALUES
```

```
 (2001, 'internal generation', 0, 0, @SYSOP, 0, 0, 0, 2001, 0),
 (0, NULL, 0, 0, @SYSOP, NULL, NULL, 0, 2001, NULL);

-- we can also amend REASONS:
ALTER TABLE REASONS ADD CONSTRAINT bad_reason_src foreign key(src)
   references SOURCES(src);
```

We insert an initial value for “internal generation”, which uses the minimum permissible key value. We also need the integral time-zone source, ***tz***:

```
INSERT INTO SOURCES
 (src, master, institution, script, amender, t_amended, reason,
  p_amended, ssource, chk)
VALUES
 (2005, 'TZ database', 0, 0, @SYSOP, 0, 0, 0, 2001, 0);
```

## 2.6 PEOPLE

Obviously the central table of our database refers to people. These people can occupy various roles, which should be checked for validity. Note that the patient occupies a very special role, in that each table that refers to patient data has a denormalised patient reference called `tag_person`. This field is used in low-level checking for an error involving patient data misattribution, and permits ready caching of data by patient.

```
SELECT Now() `Starting`,"PEOPLE" `Table`;

CREATE TABLE PEOPLE
( person integer
    ,constraint person_pk PRIMARY KEY(person)
  ,t_amended BIGINT
  ,reason int
  ,p_amended BIGINT
  ,amender int
  ,t_born BIGINT
  ,t_born_P int
  ,p_born BIGINT
  ,t_died BIGINT
  ,t_died_P int
  ,p_died BIGINT
  ,flags BIGINT
  ,srcID BIGINT
  ,src BIGINT
    ,constraint src_people foreign key(src)
       references SOURCES(src)
  ,ver integer default 0
  ,chk int
)CHARACTER SET=utf8mb4
    COLLATE=utf8mb4_unicode_ci;
```

The fields should be mostly self-explanatory. The `src` details where the data originated. A constraint on the amender will need to be inserted subsequently, as MySQL doesn't like *de novo* self-referential tables:

```
-- insert reference to system operator, code 2000.
INSERT INTO PEOPLE
  (person, t_amended, p_amended, amender, t_born, p_born, src, chk, ver)
VALUES
  (@SYSOP,   0,         0,         @SYSOP,    0,      0,      2001,  0,    0);
-- we really should insert valid times and check value here. [fix me]

INSERT INTO PEOPLE
  (person, t_amended, p_amended, amender, t_born, p_born, src, chk, ver)
VALUES
  (0,   0,  0,   @SYSOP,  0,  0,  2001, 0,  0); -- "nobody"
```

In the above, the "first user" has code 2000 (the minimum permissible value), and the src is "internal" (code zero); the "nobody" code is used to indicate missing data or a "null person" (e.g. scheduling where you cannot know the identity of the person involved).

### 2.6.1 Updates

Now that **PEOPLE** is defined and populated as above, we can add relevant constraints to other tables:

```
ALTER TABLE PEOPLE ADD CONSTRAINT bad_people_amender foreign key(amender)
  references PEOPLE(person);

ALTER TABLE Sourcescripts ADD CONSTRAINT bad_sourcescript_amender foreign key(amender)
  references PEOPLE(person);

ALTER TABLE SOURCES ADD CONSTRAINT bad_source_amender foreign key(amender)
  references PEOPLE(person);

-- for REASONS too:
ALTER TABLE REASONS ADD CONSTRAINT bad_reason_amender foreign key(amender)
  references PEOPLE(person);
```

## 2.7 PLACES

The rationale for the following fields is discussed below. This is all rather crude, I'm afraid.

```
SELECT Now() `Starting`,"PLACES" `Table`;

CREATE TABLE PLACES
( place BIGINT
    ,constraint place_pk PRIMARY KEY (place)
  ,description varchar(128)
  ,shortname varchar(32)
```

```
  ,amender int
  ,constraint bad_place_amender foreign key(amender)
      references PEOPLE(person)
  ,srcID BIGINT
  ,src BIGINT
    ,constraint src_place foreign key(src)
       references SOURCES(src)
  ,t_amended BIGINT
  ,reason int
  ,p_amended BIGINT
    -- here have relevant geospatial coordinates, with precision:
  ,east integer
  ,north integer
  ,elevation integer
  ,east_P integer -- precision
  ,north_P integer
  ,elevation_P integer
  ,ver integer default 0
  ,chk int
)CHARACTER SET=utf8mb4
    COLLATE=utf8mb4_unicode_ci;SHOW WARNINGS;DESCRIBE PLACES;
```

Precision is in nominal centimetres (at the equator). The p_amended entry is of interest, as it makes the PLACES table recursive. This evil does however allow the tricky designer (if they wish) to create a hierarchy of places, in that p_amended might contain other places.[3] A null place:

```
INSERT INTO PLACES (place, description,
  p_amended, t_amended, reason, amender, src, chk)
  VALUES (0, 'nowhere',
  0, 0, 0, @SYSOP, 2001, 0);
```

We can now add the necessary constraint:

```
ALTER TABLE PLACES ADD CONSTRAINT bad_place_place foreign key(p_amended)
  references PLACES(place);
```

We also require a base for “UTC” (universal time).

```
 insert into PLACES (place, description, amender, p_amended, src, east, north, reason)
  values (1, 'UTC', @SYSOP, 0, 0, 0, 0, 0);
```

The Perl **vector seekwell** program will also establish a **PLACES** entry that refers back to this base.

### 2.7.1 Coordinates

The Universal Transverse Mercator coordinate system (UTM) is commonly used, but this is not my choice (See below).[4] UTM specifies a zone (there

[3] This is one of the few nods made by the database to ontology.

[4] At the equator, 1.1 m precision requires GPS coordinates with seconds specified to to 5 digit precision. Although the Military grid reference system (which uses a single, compound reference) has the merit of simplicity and also describes precision, its compound nature violates one of the rules of our system.

are 60 such zones) and then two offsets in metres, first the easting, and then the northing. All coordinates are positive, due to arbitrary addition of 500 km to the central meridians and 10,000 km to the distance south of the equator in the southern hemisphere. A reference coordinate system is required — this is commonly WGS 84 (the World Geodetic System). The shape of the earth is modelled, and position is specified on this modelled shape. Note that if we wished to specify coordinates above the surface of the earth, we need a field to represent elevation.[5]

Simple use of GPS coordinates is unwise, as common queries become inefficient. For example if we ask "Who is on Ward 65?" this becomes translated into a complex query involving a range of 3-D coordinates, together with their tolerances. In addition, storage will be inefficient. Tabular abstraction seems appropriate.

### 2.7.2 My choice

Simplest is to use latitude and longitude in degrees, with altitute above WGS 84 specified as an integer value in cm. The circumference of the earth at the equator is given as 40,075,161.2m[6], so 180 degrees will correspond to ~2,003,758,060 cm. This is less than the number we can fit into $2^{31}$(a signed 32-bit integer: 2,147,483,648) so it makes sense to use signed integers to represent degrees of displacement in longitude and latitude (Potential precision will improve as we near the poles). We will convert the number of degrees to an integer by dividing by 9; multiplying by 100,187,903; and then converting to an integer by truncation of the floating point number used in calculation.[7]

For the sake of consistency, we'll use the same conversion for North-South displacements (South is negative), despite the fact that a North/South "great circle" does not have the same dimensions as the equatorial great circle (the Earth approximates an oblate spheroid).

### 2.7.3 Using Google maps

This is very convenient, especially as you can type in a pair of coordinates and Google Maps will take you there. To get a pair from Google Maps, find your location and then click on the link icon in the top right section of the browser. You will obtain a URL something like:

```
http://www.google.co.nz/mapmaker?ll=-36.861493,174.772696&spn=0.021666,0.032701
&z=15&lyt=large_map&hll=-36.860051,174.773517&hyaw=264.663684154463
```

The "ll" coordinates approximate the latitude and longitude, but may be offset. Certain websites allow direct access e.g. https://www.gps-coordinates.net/ For example, for Auckland City Hospital, we obtain S 36.86055° | E 174.76995°, and if we then type -36.86055,174.76995 into Google maps, we obtain the location. (The negative value is because we're south of the equator).

[5] GPS is currently not that hot at determining elevation.
[6] See e.g. http://en.wikipedia.org/wiki/Decimal_degrees circumference
[7] On reflection, it may be best to apply rounding here.

### 2.7.4 Conversion

The World coordinate converter can be useful (https://twcc.fr/) as it is detailed, and provides a variety of options. Also see http://boulter.com/gps/. In the above example, we obtain 60S 301203 5918276 for UTM coordinates.

### 2.7.5 Country

Rather than having a separate 'countries' table as first conceptualised, simply create **PLACES** entries. These will have a p_amended value of zero; however *regions* that will be used in time-zone representation will have countries as their parents, coded slightly irregularly in p_amended. A further irregularity is that the place codes (which range from 004–894) are technically "forbidden codes" in our standard coding structure, but correspond exactly to the ISO-3166-1 codes!

```
INSERT INTO PLACES (place, description, p_amended, t_amended, reason,
  amender, src, chk) VALUES
(4,'Afghanistan', 0, 0, 0, @SYSOP, 2001, 0),
(8,'Albania', 0, 0, 0, @SYSOP, 2001, 0),
(10,'Antarctica', 0, 0, 0, @SYSOP, 2001, 0),
(12,'Algeria', 0, 0, 0, @SYSOP, 2001, 0),
(16,'American Samoa', 0, 0, 0, @SYSOP, 2001, 0),
(20,'Andorra', 0, 0, 0, @SYSOP, 2001, 0),
(24,'Angola', 0, 0, 0, @SYSOP, 2001, 0),
(28,'Antigua and Barbuda', 0, 0, 0, @SYSOP, 2001, 0),
(31,'Azerbaijan', 0, 0, 0, @SYSOP, 2001, 0),
(32,'Argentina', 0, 0, 0, @SYSOP, 2001, 0),
(36,'Australia', 0, 0, 0, @SYSOP, 2001, 0),
(40,'Austria', 0, 0, 0, @SYSOP, 2001, 0),
(44,'Bahamas', 0, 0, 0, @SYSOP, 2001, 0),
(48,'Bahrain', 0, 0, 0, @SYSOP, 2001, 0),
(50,'Bangladesh', 0, 0, 0, @SYSOP, 2001, 0),
(51,'Armenia', 0, 0, 0, @SYSOP, 2001, 0),
(52,'Barbados', 0, 0, 0, @SYSOP, 2001, 0),
(56,'Belgium', 0, 0, 0, @SYSOP, 2001, 0),
(60,'Bermuda', 0, 0, 0, @SYSOP, 2001, 0),
(64,'Bhutan', 0, 0, 0, @SYSOP, 2001, 0),
(68,'Bolivia, Plurinational State of', 0, 0, 0, @SYSOP, 2001, 0),
(70,'Bosnia and Herzegovina', 0, 0, 0, @SYSOP, 2001, 0),
(72,'Botswana', 0, 0, 0, @SYSOP, 2001, 0),
(74,'Bouvet Island', 0, 0, 0, @SYSOP, 2001, 0),
(76,'Brazil', 0, 0, 0, @SYSOP, 2001, 0),
(84,'Belize', 0, 0, 0, @SYSOP, 2001, 0),
(86,'British Indian Ocean Territory', 0, 0, 0, @SYSOP, 2001, 0),
(90,'Solomon Islands', 0, 0, 0, @SYSOP, 2001, 0),
(92,'Virgin Islands, British', 0, 0, 0, @SYSOP, 2001, 0),
(96,'Brunei Darussalam', 0, 0, 0, @SYSOP, 2001, 0),
(100,'Bulgaria', 0, 0, 0, @SYSOP, 2001, 0),
(104,'Myanmar', 0, 0, 0, @SYSOP, 2001, 0),
```

```
(108,'Burundi', 0, 0, 0, @SYSOP, 2001, 0),
(112,'Belarus', 0, 0, 0, @SYSOP, 2001, 0),
(116,'Cambodia', 0, 0, 0, @SYSOP, 2001, 0),
(120,'Cameroon', 0, 0, 0, @SYSOP, 2001, 0),
(124,'Canada', 0, 0, 0, @SYSOP, 2001, 0),
(132,'Cape Verde', 0, 0, 0, @SYSOP, 2001, 0),
(136,'Cayman Islands', 0, 0, 0, @SYSOP, 2001, 0),
(140,'Central African Republic', 0, 0, 0, @SYSOP, 2001, 0),
(144,'Sri Lanka', 0, 0, 0, @SYSOP, 2001, 0),
(148,'Chad', 0, 0, 0, @SYSOP, 2001, 0),
(152,'Chile', 0, 0, 0, @SYSOP, 2001, 0),
(156,'China', 0, 0, 0, @SYSOP, 2001, 0),
(158,'Taiwan, Province of China', 0, 0, 0, @SYSOP, 2001, 0),
(162,'Christmas Island', 0, 0, 0, @SYSOP, 2001, 0),
(166,'Cocos (Keeling) Islands', 0, 0, 0, @SYSOP, 2001, 0),
(170,'Colombia', 0, 0, 0, @SYSOP, 2001, 0),
(174,'Comoros', 0, 0, 0, @SYSOP, 2001, 0),
(175,'Mayotte', 0, 0, 0, @SYSOP, 2001, 0),
(178,'Congo', 0, 0, 0, @SYSOP, 2001, 0),
(180,'Congo, the Democratic Republic of the', 0, 0, 0, @SYSOP, 2001, 0),
(184,'Cook Islands', 0, 0, 0, @SYSOP, 2001, 0),
(188,'Costa Rica', 0, 0, 0, @SYSOP, 2001, 0),
(191,'Croatia', 0, 0, 0, @SYSOP, 2001, 0),
(192,'Cuba', 0, 0, 0, @SYSOP, 2001, 0),
(196,'Cyprus', 0, 0, 0, @SYSOP, 2001, 0),
(203,'Czech Republic', 0, 0, 0, @SYSOP, 2001, 0),
(204,'Benin', 0, 0, 0, @SYSOP, 2001, 0),
(208,'Denmark', 0, 0, 0, @SYSOP, 2001, 0),
(212,'Dominica', 0, 0, 0, @SYSOP, 2001, 0),
(214,'Dominican Republic', 0, 0, 0, @SYSOP, 2001, 0),
(218,'Ecuador', 0, 0, 0, @SYSOP, 2001, 0),
(222,'El Salvador', 0, 0, 0, @SYSOP, 2001, 0),
(226,'Equatorial Guinea', 0, 0, 0, @SYSOP, 2001, 0),
(231,'Ethiopia', 0, 0, 0, @SYSOP, 2001, 0),
(232,'Eritrea', 0, 0, 0, @SYSOP, 2001, 0),
(233,'Estonia', 0, 0, 0, @SYSOP, 2001, 0),
(234,'Faroe Islands', 0, 0, 0, @SYSOP, 2001, 0),
(238,'Falkland Islands (Malvinas)', 0, 0, 0, @SYSOP, 2001, 0),
(239,'South Georgia and the South Sandwich Islands', 0, 0, 0, @SYSOP, 2001, 0),
(242,'Fiji', 0, 0, 0, @SYSOP, 2001, 0),
(246,'Finland', 0, 0, 0, @SYSOP, 2001, 0),
(248,'Åland Islands', 0, 0, 0, @SYSOP, 2001, 0),
(250,'France', 0, 0, 0, @SYSOP, 2001, 0),
(254,'French Guiana', 0, 0, 0, @SYSOP, 2001, 0),
(258,'French Polynesia', 0, 0, 0, @SYSOP, 2001, 0),
(260,'French Southern Territories', 0, 0, 0, @SYSOP, 2001, 0),
(262,'Djibouti', 0, 0, 0, @SYSOP, 2001, 0),
(266,'Gabon', 0, 0, 0, @SYSOP, 2001, 0),
(268,'Georgia', 0, 0, 0, @SYSOP, 2001, 0),
(270,'Gambia', 0, 0, 0, @SYSOP, 2001, 0);
```

More:

```
INSERT INTO PLACES (place, description, p_amended, t_amended, reason,
  amender, src, chk) VALUES
(275,'Palestine, State of', 0, 0, 0, @SYSOP, 2001, 0),
(276,'Germany', 0, 0, 0, @SYSOP, 2001, 0),
(288,'Ghana', 0, 0, 0, @SYSOP, 2001, 0),
(292,'Gibraltar', 0, 0, 0, @SYSOP, 2001, 0),
(296,'Kiribati', 0, 0, 0, @SYSOP, 2001, 0),
(300,'Greece', 0, 0, 0, @SYSOP, 2001, 0),
(304,'Greenland', 0, 0, 0, @SYSOP, 2001, 0),
(308,'Grenada', 0, 0, 0, @SYSOP, 2001, 0),
(312,'Guadeloupe', 0, 0, 0, @SYSOP, 2001, 0),
(316,'Guam', 0, 0, 0, @SYSOP, 2001, 0),
(320,'Guatemala', 0, 0, 0, @SYSOP, 2001, 0),
(324,'Guinea', 0, 0, 0, @SYSOP, 2001, 0),
(328,'Guyana', 0, 0, 0, @SYSOP, 2001, 0),
(332,'Haiti', 0, 0, 0, @SYSOP, 2001, 0),
(334,'Heard Island and McDonald Islands', 0, 0, 0, @SYSOP, 2001, 0),
(336,'Holy See (Vatican City State)', 0, 0, 0, @SYSOP, 2001, 0),
(340,'Honduras', 0, 0, 0, @SYSOP, 2001, 0),
(344,'Hong Kong', 0, 0, 0, @SYSOP, 2001, 0),
(348,'Hungary', 0, 0, 0, @SYSOP, 2001, 0),
(352,'Iceland', 0, 0, 0, @SYSOP, 2001, 0),
(356,'India', 0, 0, 0, @SYSOP, 2001, 0),
(360,'Indonesia', 0, 0, 0, @SYSOP, 2001, 0),
(364,'Iran, Islamic Republic of', 0, 0, 0, @SYSOP, 2001, 0),
(368,'Iraq', 0, 0, 0, @SYSOP, 2001, 0),
(372,'Ireland', 0, 0, 0, @SYSOP, 2001, 0),
(376,'Israel', 0, 0, 0, @SYSOP, 2001, 0),
(380,'Italy', 0, 0, 0, @SYSOP, 2001, 0),
(384,'Côte d''Ivoire', 0, 0, 0, @SYSOP, 2001, 0),
(388,'Jamaica', 0, 0, 0, @SYSOP, 2001, 0),
(392,'Japan', 0, 0, 0, @SYSOP, 2001, 0),
(398,'Kazakhstan', 0, 0, 0, @SYSOP, 2001, 0),
(400,'Jordan', 0, 0, 0, @SYSOP, 2001, 0),
(404,'Kenya', 0, 0, 0, @SYSOP, 2001, 0),
(408,'Korea, Democratic People''s Republic of', 0, 0, 0, @SYSOP, 2001, 0),
(410,'Korea, Republic of', 0, 0, 0, @SYSOP, 2001, 0),
(414,'Kuwait', 0, 0, 0, @SYSOP, 2001, 0),
(417,'Kyrgyzstan', 0, 0, 0, @SYSOP, 2001, 0),
(418,'Lao People''s Democratic Republic', 0, 0, 0, @SYSOP, 2001, 0),
(422,'Lebanon', 0, 0, 0, @SYSOP, 2001, 0),
(426,'Lesotho', 0, 0, 0, @SYSOP, 2001, 0),
(428,'Latvia', 0, 0, 0, @SYSOP, 2001, 0),
(430,'Liberia', 0, 0, 0, @SYSOP, 2001, 0),
(434,'Libya', 0, 0, 0, @SYSOP, 2001, 0),
(438,'Liechtenstein', 0, 0, 0, @SYSOP, 2001, 0),
(440,'Lithuania', 0, 0, 0, @SYSOP, 2001, 0),
(442,'Luxembourg', 0, 0, 0, @SYSOP, 2001, 0),
```

```
(446,'Macao', 0, 0, 0, @SYSOP, 2001, 0),
(450,'Madagascar', 0, 0, 0, @SYSOP, 2001, 0),
(454,'Malawi', 0, 0, 0, @SYSOP, 2001, 0),
(458,'Malaysia', 0, 0, 0, @SYSOP, 2001, 0),
(462,'Maldives', 0, 0, 0, @SYSOP, 2001, 0),
(466,'Mali', 0, 0, 0, @SYSOP, 2001, 0),
(470,'Malta', 0, 0, 0, @SYSOP, 2001, 0),
(474,'Martinique', 0, 0, 0, @SYSOP, 2001, 0),
(478,'Mauritania', 0, 0, 0, @SYSOP, 2001, 0),
(480,'Mauritius', 0, 0, 0, @SYSOP, 2001, 0),
(484,'Mexico', 0, 0, 0, @SYSOP, 2001, 0),
(492,'Monaco', 0, 0, 0, @SYSOP, 2001, 0),
(496,'Mongolia', 0, 0, 0, @SYSOP, 2001, 0),
(498,'Moldova, Republic of', 0, 0, 0, @SYSOP, 2001, 0),
(499,'Montenegro', 0, 0, 0, @SYSOP, 2001, 0),
(500,'Montserrat', 0, 0, 0, @SYSOP, 2001, 0),
(504,'Morocco', 0, 0, 0, @SYSOP, 2001, 0),
(508,'Mozambique', 0, 0, 0, @SYSOP, 2001, 0),
(512,'Oman', 0, 0, 0, @SYSOP, 2001, 0),
(516,'Namibia', 0, 0, 0, @SYSOP, 2001, 0),
(520,'Nauru', 0, 0, 0, @SYSOP, 2001, 0),
(524,'Nepal', 0, 0, 0, @SYSOP, 2001, 0),
(528,'Netherlands', 0, 0, 0, @SYSOP, 2001, 0);
```

Yet more:

```
INSERT INTO PLACES (place, description, p_amended, t_amended, reason,
  amender, src, chk) VALUES
(531,'Curaçao', 0, 0, 0, @SYSOP, 2001, 0),
(533,'Aruba', 0, 0, 0, @SYSOP, 2001, 0),
(534,'Sint Maarten (Dutch part)', 0, 0, 0, @SYSOP, 2001, 0),
(535,'Bonaire, Sint Eustatius and Saba', 0, 0, 0, @SYSOP, 2001, 0),
(540,'New Caledonia', 0, 0, 0, @SYSOP, 2001, 0),
(548,'Vanuatu', 0, 0, 0, @SYSOP, 2001, 0),
(554,'New Zealand', 0, 0, 0, @SYSOP, 2001, 0),
(558,'Nicaragua', 0, 0, 0, @SYSOP, 2001, 0),
(562,'Niger', 0, 0, 0, @SYSOP, 2001, 0),
(566,'Nigeria', 0, 0, 0, @SYSOP, 2001, 0),
(570,'Niue', 0, 0, 0, @SYSOP, 2001, 0),
(574,'Norfolk Island', 0, 0, 0, @SYSOP, 2001, 0),
(578,'Norway', 0, 0, 0, @SYSOP, 2001, 0),
(580,'Northern Mariana Islands', 0, 0, 0, @SYSOP, 2001, 0),
(581,'United States Minor Outlying Islands', 0, 0, 0, @SYSOP, 2001, 0),
(583,'Micronesia, Federated States of', 0, 0, 0, @SYSOP, 2001, 0),
(584,'Marshall Islands', 0, 0, 0, @SYSOP, 2001, 0),
(585,'Palau', 0, 0, 0, @SYSOP, 2001, 0),
(586,'Pakistan', 0, 0, 0, @SYSOP, 2001, 0),
(591,'Panama', 0, 0, 0, @SYSOP, 2001, 0),
(598,'Papua New Guinea', 0, 0, 0, @SYSOP, 2001, 0),
(600,'Paraguay', 0, 0, 0, @SYSOP, 2001, 0),
```

```
(604,'Peru', 0, 0, 0, @SYSOP, 2001, 0),
(608,'Philippines', 0, 0, 0, @SYSOP, 2001, 0),
(612,'Pitcairn', 0, 0, 0, @SYSOP, 2001, 0),
(616,'Poland', 0, 0, 0, @SYSOP, 2001, 0),
(620,'Portugal', 0, 0, 0, @SYSOP, 2001, 0),
(624,'Guinea-Bissau', 0, 0, 0, @SYSOP, 2001, 0),
(626,'Timor-Leste', 0, 0, 0, @SYSOP, 2001, 0),
(630,'Puerto Rico', 0, 0, 0, @SYSOP, 2001, 0),
(634,'Qatar', 0, 0, 0, @SYSOP, 2001, 0),
(638,'Réunion', 0, 0, 0, @SYSOP, 2001, 0),
(642,'Romania', 0, 0, 0, @SYSOP, 2001, 0),
(643,'Russian Federation', 0, 0, 0, @SYSOP, 2001, 0),
(646,'Rwanda', 0, 0, 0, @SYSOP, 2001, 0),
(652,'Saint Barthélemy', 0, 0, 0, @SYSOP, 2001, 0),
(654,'Saint Helena, Ascension and Tristan da Cunha', 0, 0, 0, @SYSOP, 2001, 0),
(659,'Saint Kitts and Nevis', 0, 0, 0, @SYSOP, 2001, 0),
(660,'Anguilla', 0, 0, 0, @SYSOP, 2001, 0),
(662,'Saint Lucia', 0, 0, 0, @SYSOP, 2001, 0),
(663,'Saint Martin (French part)', 0, 0, 0, @SYSOP, 2001, 0),
(666,'Saint Pierre and Miquelon', 0, 0, 0, @SYSOP, 2001, 0),
(670,'Saint Vincent and the Grenadines', 0, 0, 0, @SYSOP, 2001, 0),
(674,'San Marino', 0, 0, 0, @SYSOP, 2001, 0),
(678,'Sao Tome and Principe', 0, 0, 0, @SYSOP, 2001, 0),
(682,'Saudi Arabia', 0, 0, 0, @SYSOP, 2001, 0),
(686,'Senegal', 0, 0, 0, @SYSOP, 2001, 0),
(688,'Serbia', 0, 0, 0, @SYSOP, 2001, 0),
(690,'Seychelles', 0, 0, 0, @SYSOP, 2001, 0),
(694,'Sierra Leone', 0, 0, 0, @SYSOP, 2001, 0),
(702,'Singapore', 0, 0, 0, @SYSOP, 2001, 0),
(703,'Slovakia', 0, 0, 0, @SYSOP, 2001, 0),
(704,'Viet Nam', 0, 0, 0, @SYSOP, 2001, 0),
(705,'Slovenia', 0, 0, 0, @SYSOP, 2001, 0),
(706,'Somalia', 0, 0, 0, @SYSOP, 2001, 0),
(710,'South Africa', 0, 0, 0, @SYSOP, 2001, 0),
(716,'Zimbabwe', 0, 0, 0, @SYSOP, 2001, 0),
(724,'Spain', 0, 0, 0, @SYSOP, 2001, 0),
(728,'South Sudan', 0, 0, 0, @SYSOP, 2001, 0),
(729,'Sudan', 0, 0, 0, @SYSOP, 2001, 0),
(732,'Western Sahara', 0, 0, 0, @SYSOP, 2001, 0),
(740,'Suriname', 0, 0, 0, @SYSOP, 2001, 0),
(744,'Svalbard and Jan Mayen', 0, 0, 0, @SYSOP, 2001, 0),
(748,'Swaziland', 0, 0, 0, @SYSOP, 2001, 0),
(752,'Sweden', 0, 0, 0, @SYSOP, 2001, 0),
(756,'Switzerland', 0, 0, 0, @SYSOP, 2001, 0),
(760,'Syrian Arab Republic', 0, 0, 0, @SYSOP, 2001, 0);
```

Finally:

```
INSERT INTO PLACES (place, description, p_amended, t_amended, reason,
  amender, src, chk) VALUES
```

```
(762,'Tajikistan', 0, 0, 0, @SYSOP, 2001, 0),
(764,'Thailand', 0, 0, 0, @SYSOP, 2001, 0),
(768,'Togo', 0, 0, 0, @SYSOP, 2001, 0),
(772,'Tokelau', 0, 0, 0, @SYSOP, 2001, 0),
(776,'Tonga', 0, 0, 0, @SYSOP, 2001, 0),
(780,'Trinidad and Tobago', 0, 0, 0, @SYSOP, 2001, 0),
(784,'United Arab Emirates', 0, 0, 0, @SYSOP, 2001, 0),
(788,'Tunisia', 0, 0, 0, @SYSOP, 2001, 0),
(792,'Turkey', 0, 0, 0, @SYSOP, 2001, 0),
(795,'Turkmenistan', 0, 0, 0, @SYSOP, 2001, 0),
(796,'Turks and Caicos Islands', 0, 0, 0, @SYSOP, 2001, 0),
(798,'Tuvalu', 0, 0, 0, @SYSOP, 2001, 0),
(800,'Uganda', 0, 0, 0, @SYSOP, 2001, 0),
(804,'Ukraine', 0, 0, 0, @SYSOP, 2001, 0),
(807,'Macedonia, the former Yugoslav Republic of', 0, 0, 0, @SYSOP, 2001, 0),
(818,'Egypt', 0, 0, 0, @SYSOP, 2001, 0),
(826,'United Kingdom', 0, 0, 0, @SYSOP, 2001, 0),
(831,'Guernsey', 0, 0, 0, @SYSOP, 2001, 0),
(832,'Jersey', 0, 0, 0, @SYSOP, 2001, 0),
(833,'Isle of Man', 0, 0, 0, @SYSOP, 2001, 0),
(834,'Tanzania, United Republic of', 0, 0, 0, @SYSOP, 2001, 0),
(840,'United States', 0, 0, 0, @SYSOP, 2001, 0),
(850,'Virgin Islands, U.S.', 0, 0, 0, @SYSOP, 2001, 0),
(854,'Burkina Faso', 0, 0, 0, @SYSOP, 2001, 0),
(858,'Uruguay', 0, 0, 0, @SYSOP, 2001, 0),
(860,'Uzbekistan', 0, 0, 0, @SYSOP, 2001, 0),
(862,'Venezuela, Bolivarian Republic of', 0, 0, 0, @SYSOP, 2001, 0),
(876,'Wallis and Futuna', 0, 0, 0, @SYSOP, 2001, 0),
(882,'Samoa', 0, 0, 0, @SYSOP, 2001, 0),
(887,'Yemen', 0, 0, 0, @SYSOP, 2001, 0),
(894,'Zambia', 0, 0, 0, @SYSOP, 2001, 0);
```

### 2.7.6 Country codes

Introduced in Version 1,023,000.

```
SELECT Now() `Starting`,"countrycodes" `Table`;

CREATE TABLE countrycodes
( country BIGINT
    ,constraint cc_country FOREIGN KEY (country)
      references PLACES(place)
    ,constraint cc_pk PRIMARY KEY (country)
  ,ccode varchar(2)
  ,tla varchar(3)
  ,ver integer default 0
  ,chk int default 0
)CHARACTER SET=utf8mb4
    COLLATE=utf8mb4_unicode_ci;SHOW WARNINGS;DESCRIBE countrycodes;
```

Populate the table:

```
INSERT INTO countrycodes (ccode, tla, country)
VALUES
('XX', 'NUL', 0), -- generic null!
('UT', 'UTC', 1), -- for UTC !
('AF', 'AFG',4),   ('AX', 'ALA',248), ('AL', 'ALB',8),   ('DZ', 'DZA',12),  ('AS', 'ASM',16),
('AD', 'AND',20),  ('AO', 'AGO',24),  ('AI', 'AIA',660), ('AQ', 'ATA',10),  ('AG', 'ATG',28),
('AR', 'ARG',32),  ('AM', 'ARM',51),  ('AW', 'ABW',533), ('AU', 'AUS',36),  ('AT', 'AUT',40),
('AZ', 'AZE',31),  ('BS', 'BHS',44),  ('BH', 'BHR',48),  ('BD', 'BGD',50),  ('BB', 'BRB',52),
('BY', 'BLR',112), ('BE', 'BEL',56),  ('BZ', 'BLZ',84),  ('BJ', 'BEN',204), ('BM', 'BMU',60),
('BT', 'BTN',64),  ('BO', 'BOL',68),  ('BQ', 'BES',535), ('BA', 'BIH',70),  ('BW', 'BWA',72),
('BV', 'BVT',74),  ('BR', 'BRA',76),  ('IO', 'IOT',86),  ('BN', 'BRN',96),  ('BG', 'BGR',100),
('BF', 'BFA',854),
('BI', 'BDI',108), ('KH', 'KHM',116), ('CM', 'CMR',120), ('CA', 'CAN',124), ('CV', 'CPV',132),
('KY', 'CYM',136), ('CF', 'CAF',140), ('TD', 'TCD',148), ('CL', 'CHL',152), ('CN', 'CHN',156),
('CX', 'CXR',162), ('CC', 'CCK',166), ('CO', 'COL',170), ('KM', 'COM',174), ('CG', 'COG',178),
('CD', 'COD',180), ('CK', 'COK',184), ('CR', 'CRI',188), ('CI', 'CIV',384), ('HR', 'HRV',191),
('CU', 'CUB',192), ('CW', 'CUW',531), ('CY', 'CYP',196), ('CZ', 'CZE',203), ('DK', 'DNK',208),
('DJ', 'DJI',262), ('DM', 'DMA',212), ('DO', 'DOM',214), ('EC', 'ECU',218), ('EG', 'EGY',818),
('SV', 'SLV',222), ('GQ', 'GNQ',226), ('ER', 'ERI',232), ('EE', 'EST',233), ('ET', 'ETH',231),
('FK', 'FLK',238), ('FO', 'FRO',234), ('FJ', 'FJI',242), ('FI', 'FIN',246), ('FR', 'FRA',250),
('GF', 'GUF',254), ('PF', 'PYF',258), ('TF', 'ATF',260), ('GA', 'GAB',266), ('GM', 'GMB',270),
('GE', 'GEO',268), ('DE', 'DEU',276), ('GH', 'GHA',288), ('GI', 'GIB',292), ('GR', 'GRC',300),
('GL', 'GRL',304), ('GD', 'GRD',308), ('GP', 'GLP',312), ('GU', 'GUM',316), ('GT', 'GTM',320),
('GG', 'GGY',831), ('GN', 'GIN',324), ('GW', 'GNB',624), ('GY', 'GUY',328), ('HT', 'HTI',332),
('HM', 'HMD',334), ('VA', 'VAT',336), ('HN', 'HND',340), ('HK', 'HKG',344), ('HU', 'HUN',348),
('IS', 'ISL',352), ('IN', 'IND',356), ('ID', 'IDN',360), ('IR', 'IRN',364), ('IQ', 'IRQ',368),
('IE', 'IRL',372), ('IM', 'IMN',833), ('IL', 'ISR',376), ('IT', 'ITA',380), ('JM', 'JAM',388),
('JP', 'JPN',392), ('JE', 'JEY',832), ('JO', 'JOR',400), ('KZ', 'KAZ',398), ('KE', 'KEN',404),
('KI', 'KIR',296), ('KP', 'PRK',408), ('KR', 'KOR',410), ('KW', 'KWT',414), ('KG', 'KGZ',417),
('LA', 'LAO',418), ('LV', 'LVA',428), ('LB', 'LBN',422), ('LS', 'LSO',426), ('LR', 'LBR',430),
('LY', 'LBY',434), ('LI', 'LIE',438), ('LT', 'LTU',440), ('LU', 'LUX',442), ('MO', 'MAC',446);

INSERT INTO countrycodes (ccode, tla, country)
VALUES
('MK', 'MKD',807), ('MG', 'MDG',450), ('MW', 'MWI',454), ('MY', 'MYS',458), ('MV', 'MDV',462),
('ML', 'MLI',466), ('MT', 'MLT',470), ('MH', 'MHL',584), ('MQ', 'MTQ',474), ('MR', 'MRT',478),
('MU', 'MUS',480), ('YT', 'MYT',175), ('MX', 'MEX',484), ('FM', 'FSM',583), ('MD', 'MDA',498),
('MC', 'MCO',492), ('MN', 'MNG',496), ('ME', 'MNE',499), ('MS', 'MSR',500), ('MA', 'MAR',504),
('MZ', 'MOZ',508), ('MM', 'MMR',104), ('NA', 'NAM',516), ('NR', 'NRU',520), ('NP', 'NPL',524),
('NL', 'NLD',528), ('NC', 'NCL',540), ('NZ', 'NZL',554), ('NI', 'NIC',558), ('NE', 'NER',562),
('NG', 'NGA',566), ('NU', 'NIU',570), ('NF', 'NFK',574), ('MP', 'MNP',580), ('NO', 'NOR',578),
('OM', 'OMN',512), ('PK', 'PAK',586), ('PW', 'PLW',585), ('PS', 'PSE',275), ('PA', 'PAN',591),
('PG', 'PNG',598), ('PY', 'PRY',600), ('PE', 'PER',604), ('PH', 'PHL',608), ('PN', 'PCN',612),
('PL', 'POL',616), ('PT', 'PRT',620), ('PR', 'PRI',630), ('QA', 'QAT',634), ('RE', 'REU',638),
('RO', 'ROU',642), ('RU', 'RUS',643), ('RW', 'RWA',646), ('BL', 'BLM',652), ('SH', 'SHN',654),
('KN', 'KNA',659), ('LC', 'LCA',662), ('MF', 'MAF',663), ('PM', 'SPM',666), ('VC', 'VCT',670),
('WS', 'WSM',882), ('SM', 'SMR',674), ('ST', 'STP',678), ('SA', 'SAU',682), ('SN', 'SEN',686),
('RS', 'SRB',688), ('SC', 'SYC',690), ('SL', 'SLE',694), ('SG', 'SGP',702), ('SX', 'SXM',534),
('SK', 'SVK',703), ('SI', 'SVN',705), ('SB', 'SLB',90), ('SO', 'SOM',706), ('ZA', 'ZAF',710),
('GS', 'SGS',239), ('SS', 'SSD',728), ('ES', 'ESP',724), ('LK', 'LKA',144), ('SD', 'SDN',729),
('SR', 'SUR',740), ('SJ', 'SJM',744), ('SZ', 'SWZ',748), ('SE', 'SWE',752), ('CH', 'CHE',756),
('SY', 'SYR',760), ('TW', 'TWN',158), ('TJ', 'TJK',762), ('TZ', 'TZA',834), ('TH', 'THA',764),
('TL', 'TLS',626), ('TG', 'TGO',768), ('TK', 'TKL',772), ('TO', 'TON',776), ('TT', 'TTO',780),
('TN', 'TUN',788), ('TR', 'TUR',792), ('TM', 'TKM',795), ('TC', 'TCA',796), ('TV', 'TUV',798),
('UG', 'UGA',800), ('UA', 'UKR',804), ('AE', 'ARE',784), ('GB', 'GBR',826), ('US', 'USA',840),
('UM', 'UMI',581), ('UY', 'URY',858), ('UZ', 'UZB',860), ('VU', 'VUT',548), ('VE', 'VEN',862),
('VN', 'VNM',704), ('VG', 'VGB',92), ('VI', 'VIR',850), ('WF', 'WLF',876), ('EH', 'ESH',732),
('YE', 'YEM',887), ('ZM', 'ZMB',894), ('ZW', 'ZWE',716);
```

### 2.7.7 Regions

We will represent regions as **PLACES**, with a 'parent' country in the p_created field, as discussed above (This is relevant to the tz database).

## 2.8 Institutions

This table refers to e.g. the manufacturers of specific medicines, devices, instances of a 'thing', and of 'lots' of organisms. Clearly one manufacturer may produce many such items, and may even cross the "boundaries" of devices, drugs etc.

Rather confusingly, an entire database may also be represented as an "Institution", and then SOURCES that are dependent on it constitute individual tables (This prevents the clumsiness of having separate a separate DataBases table with a dependent DataTables table, and then being unable to reference these as a 'source').

```
SELECT Now() `Starting`,"Institutions" `Table`;

CREATE TABLE Institutions
( institution integer
    ,constraint institution_pk PRIMARY KEY(institution)
  ,international_identifier varchar(64)
  ,name varchar(128)
  ,nature int default 0
  ,site BIGINT
  ,constraint bad_institution_site foreign key(site)
     references PLACES(place)
  ,t_amended BIGINT
  ,reason int
  ,p_amended BIGINT
  ,amender int
  ,constraint bad_institution_amender foreign key(amender)
      references PEOPLE(person)
  ,srcID BIGINT
  ,src BIGINT
    ,constraint src_institution foreign key(src)
       references SOURCES(src)
  ,ver integer default 0
  ,chk int

  -- will likely need other fields here.
)CHARACTER SET=utf8mb4
    COLLATE=utf8mb4_unicode_ci;SHOW WARNINGS;DESCRIBE Institutions;
```

The **nature** field does not refer to an external table at present, but 0=not characterised, 1=is a database. Note that the `international_identifier` is an alternative (candidate) primary key. The `site` refers to the location of the institution, whereas `p_amended` is the location of the amender![8]

[8] In keeping with our usual convention of a time/location pair.

```
INSERT INTO Institutions (institution, name, international_identifier,
                    site,
                    t_amended, reason, p_amended,
                    amender, src, chk)
  VALUES (0, 'unknown', '0',
          0,
          0, 0, 0,
          @SYSOP, 2001, 0);
```

Now that we've defined Institutions, we can add a further constraint to Sources:

```
ALTER TABLE SOURCES ADD CONSTRAINT bad_source_institution
  foreign key(institution) references Institutions(institution);
```

## 2.9 Time

### 2.9.1 Daylight saving time (DST)

DST is further discussed in *seek_400.lyx* and the associated file *timely_400.lyx*, where Perl routines are provided that populate the following **timely** table. The main idea with the **timely** table is simple—we convert the weirdly complex set of rules in the IANA tz database into a simple table of transition times for each time zone! The Perl code mentioned does this tricky transformation.

```
SELECT Now() `Starting`,"timely" `Table`;

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;
```

In the above, all BIGINT time values are in microseconds. The times are Julian day values. The table will ordinarily be populated by a Perl routine that reads the TZ database. There may be multiple entries for each year, the year referring to local wall time. Within each year there is at least one entry, but if nothing exciting is going on, then this entry will merely be on 31 December at 23:59:59, and the **ignored** flag will be set. If a "real" transition coincides with this time, then 1 s is added to the real time in determining the "ignored" row, which is still inserted.

### 2.9.2 Leap seconds

```
SELECT Now() `Starting`,"leapseconds" `Table`;

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;
```

At certain unpredictable points, UTC time jumps in relation to GPS time, which ticks monotonically. So far, the jumps have always been +1s (as the Earth slows) but theoretically the jump could be negative, or even by two seconds. This table specifies the relative offset in **offset**, and the GPS and UTC times (in Julian microseconds) in the respective BIGINT variables. Note however that it's easy to convert the times by simply adding or subtracting the value in **offset** as the relevant fraction of a day. So, to convert a GPS time to a UTC time, start at the top, and if the time is greater than or equal to the value in gpstime, then *subtract* the value in **offset** (seconds). Otherwise test the next time down, until you get to the first entry; if the date is less than this entry, then add 9. Similarly, to convert UTC to GPS, conveniently use the Julian value in utctime, but *add* the offset value in seconds. In either case, divide the seconds value by 86400 to obtain a day fraction.

The leapseconds table will be populated from the TZ database by a Perl script.

### 2.9.3 PTABLES

The **PTABLES** table describes the database tables accessible to at least one usergroup. It must only be defined when all other relevant tables have been! It's important to note that the list of tables is largely reproduced in the **SKEYS** table—used for PK generation—but it seems unwise to use that table for our purposes here. **SKEYS** is also more limited, as it doesn't accommodate a number of other tables like L_ tables (if present).

```
CREATE TABLE PTABLES
( ptable integer AUTO_INCREMENT
    ,constraint bad_ptable_pk primary key(ptable)
  ,tablename varchar(64)

  ,t_amended BIGINT
  ,reason int
  ,p_amended BIGINT
    ,constraint bad_ptable_amended_at foreign key(p_amended)
       references PLACES(place)
  ,amender integer
```

```
    ,constraint bad_pillar_amender foreign key(amender)
      references PEOPLE(person)

  ,flags integer

  ,ver integer default 0
  ,chk int
) CHARACTER SET=utf8mb4
  COLLATE=utf8mb4_unicode_ci;SHOW WARNINGS;DESCRIBE PTABLES;
```

Crudely select into this table:

```
-- SET @SYSOP = 2000;

INSERT INTO PTABLES
  ( tablename, t_amended, reason, p_amended, amender)
SELECT
   T.TABLE_NAME
   ,JD(CURRENT_TIMESTAMP)
   ,0
   ,0
   ,@SYSOP
FROM INFORMATION_SCHEMA.TABLES T WHERE T.TABLE_SCHEMA = 'smalltime'
  AND T.TABLE_NAME NOT IN ('SKEYS', 'USERS', 'PCOLUMNS', 'PTABLES', 'PERMITS')
  AND T.TABLE_NAME NOT LIKE 'E\_%';
```

### 2.9.4 PCOLUMNS

The **PCOLUMS** table describes the database columns for the tables in PTABLES.

```
CREATE TABLE PCOLUMNS
( pcolumn integer AUTO_INCREMENT
    ,constraint bad_pcolumn_pk primary key(pcolumn)
  ,ptable integer
    ,constraint bad_pcolumn_table foreign key(ptable)
       references PTABLES(ptable)
  ,colname varchar(64)
  ,canonical_position integer
  ,column_type varchar(1) -- I J T F not varchar(64)
  ,maximum_length integer

  ,t_amended BIGINT
  ,reason int
  ,p_amended BIGINT
    ,constraint bad_pcolumn_amended_at foreign key(p_amended)
       references PLACES(place)
  ,amender integer
    ,constraint bad_pcolumn_amender foreign key(amender)
      references PEOPLE(person)
```

```
  ,ver integer default 0
  ,chk int
) CHARACTER SET=utf8mb4
  COLLATE=utf8mb4_unicode_ci;SHOW WARNINGS;DESCRIBE PCOLUMNS;
```

Select in all of the columns that are relevant:

```
-- SET @SYSOP = 2000;

INSERT INTO PCOLUMNS
  ( ptable, colname, canonical_position, column_type, maximum_length,
    t_amended, reason, p_amended, amender )
SELECT
  D.ptable
  ,C.COLUMN_NAME
  ,C.ORDINAL_POSITION
  ,CASE
    WHEN C.COLUMN_TYPE LIKE 'varchar%' THEN 'T'
    WHEN C.COLUMN_TYPE LIKE 'bigint%' THEN 'I'
    WHEN C.COLUMN_TYPE LIKE 'int%' THEN 'J'
    WHEN C.COLUMN_TYPE LIKE 'float%' THEN 'F'
    ELSE 'Q'
  END
  ,C.CHARACTER_MAXIMUM_LENGTH
  ,JD(CURRENT_TIMESTAMP)
  ,0
  ,0
  ,@SYSOP
FROM INFORMATION_SCHEMA.COLUMNS C
  INNER JOIN PTABLES D ON C.TABLE_NAME = D.tablename
WHERE C.TABLE_SCHEMA = 'smalltime'
  AND C.COLUMN_NAME NOT IN ( 'ver', 'chk' )
  AND D.tablename NOT IN ( 'skeys', 'usergroups', 'users' );
```

## 2.10 Special tz tables

***Create*** the following ancillary rule tables in ***f*ehr**. There is minimal processing, and the tables are denormalized with respect to rule_name. The sole intent of these tables is to permit reconciliation of rules when a new tz version is encountered. They are *not* used to perform calculations, but to record what was used by the *timely.pm* module in translating tz rules into entries in the **timely** table.

### 2.10.1 rule sets:

```
SELECT Now() `Starting`,"tz_rules" `Table`;

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;
```

### 2.10.2 zone cutoff rules

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:

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

```
SELECT Now() `Starting`,"tz_cutoffs" `Table`;

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;
```

We also need source rows in **SKEYS**:

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

# 3.0 ODBC connection to *smalltime*

The following describes a typical Windows connection. For Linux, see Section 3.1.

1. Download MySQL Connector/ODBC Install using the 'typical' installation configuration.
2. Open a Windows command prompt ***as administrator*** and say:

   ```
   C:\Windows\System32\odbcad32.exe
   ```

   (a) Make sure you're on the System DSN tab

   (b) Note that other ways of doing this may fail in Windows 11 (even if you select run as admin) or succeed intermittently. Also see: https://www.tech2geek.net/how-to-fix-odbc-settings-not-saving-in-windows-11-pro/
3. Click the [Add ] button
4. Select the MySQL ODBC Unicode driver, and click [Finish];
5. Enter SMALLTIME as the Data Source Name, inserting the relevant user name and *password*

**Data Source Name:** SMALLTIME

**Description:** ***smalltime*** data (or whatever)

**TCP/IP server:** selected

**Port:** 3306

**User:** A specific user (or root, if you want to be dangerous)

**Database:** all databases should pop up! (Make sure WampServer is running and 'green'). Select ***smalltime*** — you can now test the connection using the [Test] button next to the database name.

## 3.1 Linux

This can be more challenging than with Windows. In the following we go through the whole gamut, and build the ODBC driver—but I should explore other database connection options that fill in for ODBC.

### 3.1.1 Install MySQL / MariaDB

See https://www.digitalocean.com/community/tutorials/how-to-install-mariadb-on-ubuntu-22-04

```
sudo apt install mariadb-server
sudo mysql_secure_installation
```

... follow the instructions, set your password. Then:

### 3.1.2 Make a separate admin account:

```
sudo mariadb
GRANT ALL ON *.* TO 'admin'@'localhost' IDENTIFIED BY 'password'
WITH GRANT OPTION;
FLUSH PRIVILEGES;
exit
```

Check status:

```
sudo systemctl status mariadb
```

## to start would say: ## sudo systemctl start mariadb
## Now check mysqladmin:

```
sudo mysqladmin version
```

### 3.1.3 Install PhpMyAdmin

```
sudo apt install phpmyadmin
```

#THIS WILL INSTALL APACHE
You may next want to visit: http://localhost/ ; Then say:

```
sudo nano /etc/apache2/apache2.conf
```

Modify by adding this:

```
# Modified by JvS on 2025-12-17
Include /etc/phpmyadmin/apache.conf
# end modification.
sudo service apache2 restart
```

# then navigate to
http://localhost/phpmyadmin/
Login as admin (rather than root)

### 3.1.4 Make vanilla user

Login to http://localhost/phpmyadmin/ as admin
User accounts | Add user account ...
User name: vanilla
localhost
[password]
Limit privileges to SELECT, INSERT, UPDATE and DELETE
(We need DELETE for the initial setup of smalltime, but after this wise to remove this privilege).

### 3.1.5 Make your actual database

Do this, as noted in Chapter 2.

### 3.1.6 MariaDB connector

https://mariadb.com/docs/connectors/connectors-quickstart-guides/connector-odbc-guide

```
sudo apt update
sudo apt upgrade
sudo apt install unixodbc unixodbc-dev
```

You may encounter **pain** NB. get the right version. Check your Debian version. ** The following are examples only**

Go to: https://mariadb.com/downloads/connectors/connectors-data-access/odbc-connector/

Download ODBC connector | version 3.2.7-GA for Debian 12 64 bit
This will go to e.g. /home/jo/Downloads/

```
sudo mv Downloads/maria*.deb /usr/src
cd /usr/src/
ls
sudo dpkg -i mariadb-connector-odbc_3.2.7-1+maria~bookworm_amd64.deb
```

You may need `sudo apt --fix-broken install` rather a lot.

### 3.1.7 Configure odbcinst.ini and odbc.ini

Look for driver definition in */etc/odbcinst.ini*

1. For user DSNs, you need to create/modify *~/.odbc.ini*
2. For system DSNs, you must create/modify */etc/odbc.ini*

First, find the driver/setup:

```
ls -la /usr/lib/x86_64-linux-gnu/ | grep odbc
```

Look for libmaodbc.so

Open the file */etc/odbcinst.ini* and edit (sudo) the driver definition along the lines of:

```
[MariaDB_ODBC_Driver]
Description = MariaDB Connector/ODBC
Driver = /usr/lib/x86_64-linux-gnu/libmaodbc.so
Setup = /usr/lib/x86_64-linux-gnu/libmaodbc.so
UsageCount = 1
```

Similarly edit *odbc.ini* which provides the DSN, and store it in *~/.odbc.ini*

```
[SMALLTIME]
Description = My MariaDB Database
Driver = MariaDB_ODBC_Driver
SERVER = 127.0.0.1
PORT = 3306
DATABASE = smalltime
User = admin
Password = ***yourpasswordgoeshere***
```

Permissible variation:
You can change 127.0.0.1 to localhost instead.
Don't use admin as the user. Rather use an under-privileged vanilla account!

### 3.1.8 TEST:

```
isql SMALLTIME your_username your_password
```

If this fails, check dependencies:

```
ldd /usr/lib/x86_64-linux-gnu/libmaodbc.so
```

Make sure that e.g. MariaDB_ODBC_Driver is identically represented as the Driver in SMALLTIME.
## Check drivers: odbcinst -q -d and then odbcinst -q -s
## Should list the name of the driver ; and the data source.
## (may need to install odbcinst )
Use stat /dtc/odbcinst.ini to check the permissions, which should be 644. Otherwise might need to chmod.

# 4.0 The Perl script

## 4.1 The main file

This is *perl/small.pl*. It continues in Chapter 5. If you've established the desired directories e.g. *~/smalltime* and *~/smalltime/perl*, to run the program in in Linux say:

```
cd ~/smalltime/perl
perl small.pl
```

The command line menu should appear. The first time, you'll say `r all` to read the entire source from the IANA tz files you've put in *~/smalltime/perl/tz/*. (See Section 5.6 for details). Windows is similar. Most powerful is then `v all` as this validates all transitions against the timezone data installed in the Perl module. Not that particularly on Windows, the Perl version may not be up to date, resulting in discrepancies.

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

# check the following:
use feature 'unicode_strings';
use utf8;
use open ':encoding(utf8)';
binmode(STDOUT, ":utf8");

use constant LEAPSECONDCOUNT => 27;  # see usage below
my $TOPCOUNTRY = 999;           # max in PLACES table. *DO NOT* fiddle!
                                # don't make it a constant (interpolated).

my $FULLDEBUG = 0; # [nasty]
my $TOPYEAR = 2035;
my $LOWYEAR = 1895;
my($PRETTYDOTS) = 60;  # count of dots on console before newline
my $TICK  = "✔"; # or '_/' if no Utf8.
my $MAXOOPS = 100;              # maximum errors. See usage.
my $READKEYBLOCKINGPROBLEM = 0; # A problem with Perl upgrade to 5.28 !
```

```
my $TZDIR = 'tz';       # sub-directory that contains tz data files

package main;
## use Time::HiRes qw(gettimeofday usleep);  # high-resolution timing
## use POSIX qw(floor ceil);
## use File::Copy qw(move); # more 'compatible' than rename

use Term::ReadKey;  # only used to interrupt long loops by pressing Esc
$| = 1;             # turn on autoflushing

## ensure lib/Timely module is visible: [??? is this necessary ???]
use FindBin;
use lib "$FindBin::Bin/lib";

use Timely;
```

In the following, `@OLDMENUVALUES` is used to retain these attributes and access them when appropriate—only used in checking whether various menu parameters have changed, and emboldening the displayed text accordingly.

```
            # NAMES: ('region', 'placename', 'INTERVAL', 'utc', 'DayNumber',
            #         'sec', 'wall', 'myz', 'mydst');
 my(@OLDMENUVALUES) = ('',       '',          '',         '',    '',
                       '',    '',     '',    ''     );
```

Some important constants.

```
use constant ZoneReadFailed => 1000; # arbitrary failure code (should be > 100)

  ## COLOURS: 30–37 = black, red, green, yellow, blue, magenta, cyan, white.
use constant ANSIBLACK => 30;
use constant ANSIRED => 31;
use constant ANSIGREEN => 32;
use constant ANSIYELLOW => 33;
use constant ANSIBLUE => 34;
use constant ANSIMAGENTA => 35;
use constant ANSICYAN => 36;
use constant ANSIWHITE => 37;

use constant MAXZOFF => 1+14*3600;  # ) maximum ZONE offset (+14 for Kiritimati);
use constant MAXDST => 1+12*3600;   # ) seconds
use constant MINIMUMDATE => 2378496*86400;    # < 1800 is just silly

  ## convenient constants used in tz rule translation:
use constant  DASHRULE => '-';
use constant  FIXEDDST => 'd';
use constant  FIXEDDZ => 'z';
use constant  DSTTRAN => 'x';
```

```
use constant  FINALRULE => 'y';

use constant EPSILONSECONDS => 0.010;     # [fix this up cf below. 10 ms. ]

  ## ERROR codes:
use constant OnlyHelp => 1;
use constant BadTopYear => 2;

my $LEAPSECONDS = LEAPSECONDCOUNT;
my $BASEDATABASE = 'SMALLTIME';    # the target database *written to*
                                   # for change specify db=foo on the command line.
  my $TZDATABASEID = 0;              # a source (src) ID
  my $USERID = 2000;               # 'standard' default main user

my @STORAGE;
my %LOCALS = ();                   # current set of locals (empty)
my %ZONECODES;                     #  internal database zone codes
my %REZONE;                        # the reverse lookup of ZONECODES
my %RULES;                         # associative array for rules
my %ZONES;                         # and for zones
my %LINKS;                         # time zone links e.g. Europe/Jersey

 use DBI;
 my @driver_names = DBI->available_drivers;
 print ("Drivers for DBI: @driver_names \n");
 my @data_sources = DBI->data_sources('ODBC');
 print("DBI data sources for ODBC are @data_sources \n");

my $BUG = 0;                     # debugging, See Debug_()


  my $WINDOWS = 0; # signal we're NOT on Windows [explore DBI use]

  # Detect OS:
  my $OPERATING_SYSTEM;
if($^O =~ /darwin/i)
  { $OPERATING_SYSTEM = 'Mac';
  }
elsif($^O =~ /linux/i)
  { $OPERATING_SYSTEM = 'Linux';
  }
elsif($^O =~ /win/i)
  { $OPERATING_SYSTEM = 'Windows';
    $WINDOWS = 1;
    # hack: change console mode:
    `chcp 65001`;
  } else
  { $OPERATING_SYSTEM = 'Unknown'; # perhaps fail at this point!
    &DoWarn(3, " UNKNOWN OPERATING SYSTEM: '$^O'");
  };
```

```
  &PreArgs(); # read command line arguments
```

## 4.2 Connect to *smalltime*

The undef values are needed for username/password, and assume that a relevant, 'automated' database login has been setup.

```
my %DATABASES;
my $smalltime_h;
  print "\nConnecting to '$BASEDATABASE'\n";
  $smalltime_h = DBI->connect("dbi:ODBC:$BASEDATABASE",
                                  undef,
                                  undef,
                                  { AutoCommit => 0,
                                    RaiseError => 1,
                                    LongReadLen => 50000000});
if(! defined $smalltime_h)
  { die "*CRASH* could not make ODBC instance for $BASEDATABASE\n";
  };
  $smalltime_h->{LongTruncOk} = 0;
  $smalltime_h->{LongReadLen} = 100000000; # 100M !
  print "\n\n\nThe AutoCommit attribute is: " . $smalltime_h->{AutoCommit} . "\n";
  print "Maximum read length is: " . $smalltime_h->{LongReadLen} . "\n";
  print "The value of LongTruncOk is: " . $smalltime_h->{LongTruncOk} . "\n\n\n";
    # cf. https://docstore.mik.ua/orelly/linux/dbi/ch06_02.htm
```

Stuff:

```
  $DATABASES{$BASEDATABASE} = $smalltime_h; # note this is case-sensitive.
  my(@LEAPDATA) = Timely::BeginTimely($smalltime_h, 1);    # open log. [2nd argument is

if(scalar @LEAPDATA < $LEAPSECONDS)
  { &PopulateLeapSeconds($TZDIR); # fix or die.
    @LEAPDATA = Timely::BeginTimely($smalltime_h, 1);
  if(scalar @LEAPDATA < $LEAPSECONDS)
    { die "Unable to update leap seconds <@LEAPDATA>";
    };
  };
  print "\n Leap data length: " . scalar @LEAPDATA;

  my($region, $regionNAME, $hits) = Timely::FindRegion('UTC/UTC');
  Timely::SetUtcCode($region); #$UTCCODE

&ClearNewWarnings();
Timely::SetMaxKeyFetch(2500000);          # 2.5M new keys [explore]
Timely::SetUserId($USERID);
Timely::SetTzDatabase($TZDATABASEID);
```

## 4.3 Run timely

This was previously a module invocation, Timely::Timely(); now we simply invoke Timely (5.1).

```
my($fail) = Timely();
```

## 4.4 Done

```
&ImplementExit($fail);  # this should COMMIT unless $fail is nonzero.


################################################
#              END OF MAIN ROUTINE             #
################################################
```

# 5.0 Key Perl routines

This imports the UI from the timely module, and also has a few utility functions.

## 5.1 Timely

Check our date routines are functional using TestJulian; then invoke the main, interactive routine.

```
sub Timely #
{ my($fail);
  $fail = &TestJulian();  # check Julian_day number conversions are working
if($fail)
  { return($fail);
  };
  return( &Converter() );  # USER INTERFACE
}
```

Must return 0 on success; or a non-zero failure code on failure.

## 5.2 Read Time Zones

This section allows a user to interact with a diagnostic program from the command line. The menu is manufactured by SetMenu (5.4.6). Various tests are made, including checks against the Perl DateTime module, which is built around the tz database and program. First, it's important to be able to find a named timezone (region). The following summarises the help available from the command line via the menu, and by saying **h** :

**f** finds a zone based on a partial name match. The first match is returned, currently with no warning about duplicates :(

**a** gives *all* the transition times *around* the current zone.

The following timestamp setting/checking options are useful. They all set the current time too.

**u** Enter UTC-based date (actual internal representation is as a proleptic GPS date, press Enter to obtain the delta in seconds, and the corresponding wall time for the currently selected zone, default is Auckland).

**g** Enter GPS timestamp (proleptic) and obtain corresponding UTC value as Gregorian YYYY-MM- etc.

**j** Enter wall timestamp for this region (YYYY-MM-etc) and get back Julian value for GPS timestamp.

**n** Now: get the current time.

The g and j commands are complementary, for example if in Auckland you say `j 2020-01-01 12:59:42`, then you obtain a Julian value of 2458849.5 reflecting 'proleptic GPS time'. This corresponds to UTC of 2019-12-31 23:59:42, i.e. there is a difference of 18 s between UTC and GPS for this time. If instead I say `g 2458849.5` I get the same result. In addition we have a variety of tests:

**t** Test a given time for the current region, across all years, e.g. `t 01-01 00:00:00` ; you can limit the check to *after* a specific year by specifying the year too. The value entered is UTC based. See ZoneTest (5.10).

**w** Similar to **t**, but uses wall time.

**v** validates all transition times against tz. The default is to test just the current zone. More exciting is `v all` which does this for every single time zone, a great way of picking out anomalies. You can even add to or subtract from the transition point, using say v+1 or v-1.

Testing at intervals may be useful, but doesn't do an external check against tz dates—internal validation is the only test. [explain]

**i** Sets the interval in seconds. This will take forever with a small interval.

**y** Specify the year to check at that interval

Pressing the Enter key alone will simply re-display the Timely help menu.

Daylight saving is complex. In spring, the local time leaps forward at a particular time, creating a gap (invalid local times during the transition interval); in autumn the local time falls back, creating non-unique Gregorian timestamps. There are also zone changes. In ***f*ehr**, I use proleptic GPS timestamps, specified in a Julian form as a big integer (microseconds).

It's convenient to have the ability to reference not only the standard TZ time zones, but also UTC.[1]

## 5.3 Countries & regions

In ***f*ehr**, countries and regions are simply **PLACES**. Country codes are kept in the **countrycodes** table.

For each country code, the rules have varied in the past, generally by year; the rules also vary by *parts* of many countries, for example the USA and Australia, and even within parts of states (the Navajo nation). Note

[1] See http://cldr.unicode.org/development/development-process/design-proposals/extended-windows-olson-zid-mapping for Windows/Olson mapping.

that although **PLACES** does not *mandate* a reference to a particular time zone, the p_amended field can be used to signify a zone.

A similar relationship is established within the **L_location** table at the discretion of the user, who puts in a **p_amended** entry to signify a given region. This region code is then used to identify a time zone and daylight saving for that **L_location** entry.

### 5.3.1 Leap seconds

The rules and the **leapseconds** table are as specified above in Chapter 2.

### 5.3.2 Country codes

These map countries to two- and three-letter country codes.

### 5.3.3 DST dates

In the TZ database, rules for DST dates are often related to e.g. "The first Sunday in October", or whatever, rather than a given calendar date such as 7 October. For the various countries, we effectively have pairs of dates between which a value must be subtracted in converting from local time to GPS time. This offset too will vary across the years; there are also some anomalies.

Internally we record timestamps using GPS time. However, users who input dates will generally input local timestamps. They will not generally be aware of ambiguities or time skips.

## 5.4 Converter

This subroutine prints to the console cf. SetMenu (5.4.6), interacting with the user to allow conversion to and from Julian, as well as providing user control for re-reading the database and running the main tests. It loops around until the user chooses to exit.

This routine must return a non-zero value on failure, 0 for success.[2]

```
sub Converter
{
  ## &Cls();  ## temp removal 2021-03-13

  my($INTERVAL) = 3000; # seconds, a bit < 1 hour.
  my($NUMBR) = Timely::GpsJulian(2000,1,1, 0,0,0, 0, 0, 0); # seconds!!
  # GPS time is here e.g. 13s ahead of UTC
  my($region, $regionNAME, $hits) = Timely::FindRegion('Auckland'); # [parochial]
  Timely::SetRegion($region);

  my($MENU) = &SetMenu($NUMBR, $region, $INTERVAL, $regionNAME, 1); # 1=Welcome
  print $MENU;
```

The main loop. First, `r` to read. The wrinkle here is that selecting this option sets the `$confirm` flag, which requires y/yes to actually start the protracted ReadZones (5.7.2) process.

[2] I wrote it yonks ago, and it's pretty damn clumsy.

```
  my($confirm) = 0;
AGAIN: while( <STDIN> )
  { chomp;
  if($confirm)
    { $confirm = 0;
    if( /^y/i )
      { my($fail);
        $fail = &ReadZones(0, '');  # read all zones [at present] 0 signals this...
        ## ^ best should simply return failure code here if fails:
      if(length $fail > 0)  # expected null string
        { print "\n*FAILED* '$fail'\n";
          return(ZoneReadFailed);
        };
      };
      print $MENU;
      next AGAIN;
    };
  if( /^r\s*(.*?)\s*$/i ) # READ
    { my($zpart) = $1;
    if(length $zpart < 1)
      { my($fail) = &ReadZones($region, $regionNAME); # region limited.
      if(length $fail > 0)
        { print "\nOops! '$fail'\n"; # the error message.
        } else
        { print "\nCompleted zone read: '$regionNAME'.\n";
        };
      }
    elsif($zpart eq 'all') # only this will do
      { print "\n Read ENTIRE TZ database? Are you sure? (y|n) ";
        $confirm = 1;
      }
    else
      { print "\nUnknown read option '$zpart'\n";
      };
      next AGAIN;
    };
```

### 5.4.1 The rest

Test for other commands. The list of options is contained in the regex [adf... and resolved in the subsequent code. Options that are currently unused are: `b c e k m o p x`. If the user presses Enter, then the help menu will simply be displayed without any verbiage. The multiple uses of UpdateMenu (5.4.2) are a convenience.

```
  if( ! /^\s*([adfghijlnqstuvwyz\+\-\?])\s*(.*?)\s*$/i ) # 'r' already processed
    {
    if(length $_ > 0)
      { &Cls();
        print " Unknown option ($_)\n";
      } else
```

```
    { $MENU = &UpdateMenu($NUMBR, $region, $INTERVAL, $regionNAME);
    };
    next AGAIN;
  } else
  { my($inp) = $2;
    my($ch)  = $1;
    $ch =~ tr/A-Z/a-z/;  # lower case
  if($FULLDEBUG)
    { print "\n$inp--> ";
    };
```

#### 5.4.1.1 Simple

Several small commands: q h t and w follow.

```
if( $ch eq 'q' )    # Quit
  { return(0);     # success. Will force COMMIT by caller!
  };
```

h is for Help:

```
if( ( $ch eq 'h' )    # help [might make this more fancy?]
  ||( $ch eq '?'  )
  )
  { my($hlp) = &ShowHelp($inp); # returns '' on success!
  if(length $hlp > 0) # failed
    { # $MENU = &UpdateMenu($NUMBR, $region, $INTERVAL, $regionNAME);
      print $MENU;
      print $hlp;
    };
    next AGAIN;
  };
```

t is for Test:

```
if( $ch eq 't' ) # TEST
  { &ZoneTest($inp, $NUMBR, $LOWYEAR, $TOPYEAR); # global source
    print $MENU;
    next AGAIN;
  };
```

w is for wall-time test:

```
if( $ch eq 'w' ) # WALL TEST
  { &WallTest($inp, $NUMBR, $LOWYEAR, $TOPYEAR-1);
    print $MENU;
    next AGAIN;
  };
```

#### 5.4.1.2 More

Enter Gregorian date. The Gregorian date is adjusted according to the local settings to a Julian value in internal database format. It is possible that the submitted Gregorian date may be invalid. In addition to returning the Julian day, InternalJulian () checks the value. More invocation of UpdateMenu (5.4.2).

```
if($ch eq 'g')  # convert TO Julian
  {
    my($n) = Timely::InternalJulian($inp, $region, $regionNAME);
  if($n)
    { $NUMBR = $n;
      $MENU = &UpdateMenu($NUMBR, $region, $INTERVAL, $regionNAME);
      next AGAIN;
    };
  }
```

The converse, Julian to Gregorian. The Julian value is always stored in the internal format of the database, ie. proleptic GPS time.

```
 elsif($ch eq 'j' ) # TO Gregorian wall time
  { # here might further check $inp, but simply do:
  if($inp =~ /^\s*(\d+\.?\d*)\s*$/ )
    { my($JD) = $1;
    if($JD > 3000000)
      { print "A bit too late!\n";
        next AGAIN;
      };
      $inp = $JD*86400; # convert to seconds
      my($YYYY,$MM,$DD,$h,$m,$s, $dz, $dd, $hog, $deltz, $delds)
                          # unused ^    ^    ^     ^       ^
            = Timely::J2G(1, $region, $inp); # dz,dd are dummy
    if($YYYY)
      { $NUMBR = $inp;
        $MENU = &UpdateMenu($NUMBR, $region, $INTERVAL, $regionNAME);
        next AGAIN;
      };
    };
    print "Bad Julian '$inp'\n";
  }
```

Use ‘u’ like ’j’, but the conversion to Julian is based on a UTC timestamp, not a local one:

```
elsif($ch eq 'u')  # UTC to Julian
  { my ($utc) = GetUtcCode();
    my($n) = Timely::InternalJulian($inp, $utc, 'UTC/UTC');
  if($n)
    { $NUMBR = $n;
      $MENU = &UpdateMenu($NUMBR, $region, $INTERVAL, $regionNAME);
```

```
      next AGAIN;
    };
  }
```

With 'v', can check transition times for one or all zones:

```
elsif($ch eq 'v')  #
  {
    &Multitest($inp, $region); # []
  }
```

Code 'n' signals "Set current time":

```
elsif($ch eq 'n')  # get current time, put into $NUMBR
  { my($now) = Timely::ModTimeNow();
    $NUMBR = Timely::JulianFromUnix($now); # seconds
    $MENU = &UpdateMenu($NUMBR, $region, $INTERVAL, $regionNAME);
    next AGAIN;
  }
```

Find a region code (internal) based on a partial search on the name, returning first match. Again, UpdateMenu (5.4.2). NB THERE IS NO CHECKING FOR SQL injection!

```
 elsif($ch eq 'f' ) # inp is target
  {
  if($inp =~ /^\s*$/ )
    { print "Current region: $region $regionNAME\n";
    } else
    { my($rgn, $desc, $hits) = Timely::FindRegion($inp);
    if($hits < 1)
      { print "No match '$inp', nothing changed\n";
      } elsif($hits > 1)
      { print "Multiple ($hits) matches! Refine your specification.\n"
      } else
      {
      if($FULLDEBUG)
        { print "Match set to '$desc', code=$rgn\n";
        };
        Timely::SetRegion($rgn);
        $region = $rgn;      #
        $regionNAME = $desc; #
        $MENU = &UpdateMenu($NUMBR, $region, $INTERVAL, $regionNAME);
      };
    };
  }
```

Set the testing interval for the current zone (See also 'y')

```
  elsif($ch eq 'i' ) # interval in seconds
   {
```

```
  if($inp !~ /^\s*(\d+)$/)
    { print " Incorrect interval(seconds)\n";
    } else
    { $INTERVAL = $1;
      print " Interval set to $INTERVAL seconds\n";
    };
  }
```

For one zone, test a given year (at the current interval):

```
 elsif($ch eq 'y' ) # year : or year range
  {
    &TestOneZone($region, $inp, $INTERVAL); # assumes MYTIMELY already set up
  }
```

Test year (or range) with benchmark comparison (validation):

```
 elsif($ch eq 'z' ) # year : or year range
  {
    &ReconcileOneZone($region, $regionNAME, $inp, $INTERVAL); #
  }
```

Display transitions around a particular year or range of years: ‘a’:

```
 elsif($ch eq 'a' ) # year
  { my($txt) = &Around($region, $inp, $NUMBR);
  if(length $txt > 0)
    { print $txt;
    };
    next AGAIN;
  }
```

Plus and minus, with UpdateMenu (5.4.2) once more.

```
elsif( ($ch eq '+')
     ||($ch eq '-')
     )
  { my($sgn) = 1;
  if($ch eq '-')
    { $sgn = -1;
    };
    my ($k) = &TimeInSeconds($sgn, $inp); # returns zero on failure.
  if($k)
    {
    if(abs($k) < 5000000000)
      { # print "Debug: $k $sgn " . $NUMBR . "\n" ;
        $NUMBR += $k;
        $MENU = &UpdateMenu($NUMBR, $region, $INTERVAL, $regionNAME);
      } else
      { print "Perhaps $k seconds is a shade too large?\n";
      };
    };
    next AGAIN;
  }
```

List leapseconds:

```
    elsif($ch eq 'l' ) # year
     { &ListLeapseconds();
       next AGAIN;
     }
```

Done:

```
      else  # braces + belt.
      { print "\n Woops! unknown command '$ch'\n";
      };
    };
  };
}
```

End of that nasty big loop.

### 5.4.2 Update Menu

Update menu with supplied values. A convenience. The Julian number in `$NUMBR` is in seconds, not days. SetMenu (5.4.6) rejigs the menu.

```
sub UpdateMenu #
{ my($NUMBR, $region, $INTERVAL, $regionNAME) = @_;

  my($MENU) = &SetMenu($NUMBR, $region, $INTERVAL, $regionNAME, 0);
  print $MENU;
  return($MENU);
}
```

### 5.4.3 Time in seconds

Options are e.g. 60 and hh:mm:ss (no truncated form) such as 00:10:30. The sign is a text '+' or '-'

```
sub TimeInSeconds #
{ my($sgn, $txt) = @_;

  # also allow d[ays]
if($txt =~ /^\s*(\d+\.?\d*)\s*da?y?s?/ ) # [hack]
  { return( int($sgn * $1*86400) );
  };

  my($I);
if( $txt =~ /^\s*(\d+)\s*$/ )
  { return($sgn * $1);
  };   # [ might have magnitude checks ?]
if( $txt =~ /^\s*(\d+):(\d{2}):(\d{2})\s*$/ )
        # allow eg. 3:00:00 or even 12345:99:99 ! [might constrain further??]
  { return( $sgn * ($3 + 60*$2 + 3600*$1) );
  };
```

```
  print "Invalid time '$txt', enter either a number or hh:mm:ss e.g. 00:40:30\n";
  return(0);
}
```

### 5.4.4 Show help

Help with the more complex commands: t w v y z

```
sub ShowHelp #
{ my($qry) = @_;

if( $qry =~ /^elp(.+)/i )  # accommodate help or HELP etc.
  { $qry = $1;
  };

  my($h);
if( $qry =~ /^\s*(frill)/ )
  { # special
  }
elsif($qry !~ /^\s*([rtvwyz])/)
  { return("  Help options are  r t v w y z  &  frills. "
           . "Try:  h t  OR:  h frills\n"); # standard failure
  };
  $h = $1;
```

Multiple options, starting with read & test:

```
  my($txt);
if( $h eq 'r')
  { $txt = "Read 'raw' tz source data. Options are: \n"
         . "\n"
         . "  r all   Read, APPLY AND REWRITE the entire data set!\n"
         . "          **use with appropriate caution**\n"
         . "          (seekwell/perl/tz must contain tz source files)\n"
         . "\n"
         . "  r       On its own 'r' reads data for the current zone alone.\n"
         . "          This is a 'fake' read---nothing is changed.\n";
  }
elsif( $h eq 't')
  { $txt = "Test a specific month, day and UTC time for ALL zones and years!\n"
         . "  With no parameters, this defaults to the current menu time. \n"
         . "\n"
         . "  Entered values also work e.g.   t 07-01 00:00:00 \n"
         . "  as well as the lowest year:     t 2000-07-01 00:00:00\n"
         . "  or even a RANGE of years:       t 1990 2000-07-01 00:00:00\n"
         . "  Default time, between years:    t 1990 2000\n"
         . "\n"
         . "  The process FOR EACH timestamp FOR EACH zone is:\n"
         . "    1. Use Timely to convert UTC to zone-specific Gregorian\n"
         . "    2. Also convert UTC value to local Gregorian using tz\n"
```

```
        . "    3. If they match, print '.', otherwise issue warning.\n"
        . '';
  }
```

Then validate and wall test:

```
elsif( $h eq 'v')
  { $txt = "On its own, v validates all database transition times for this zone.\n"
        . "\n"
        . "  If you want to laboriously test all zones, say:   v all\n"
        . "  You can even test transition times with an offset. \n"
        . "  Try for example:   v+1   OR   v-1 \n"
        . "  Even constructs like:   v+3600 all   will work \n"
        . "\n"
        . "  The details are interesting: the Julian transition is retrieved,\n"
        . "  and converted to Gregorian for this zone, both in Timely and tz. \n"
        . "  A warning is issued if the two don't match. Note that this is NOT\n"
        . "  a guarantee that the transition point recorded in the database is\n"
        . "  correct, merely that the two wall timestamps ultimately match.\n"
        . '';
  }
elsif( $h eq 'w')
  { $txt = "This is similar to 't' but instead uses wall time. \n"
        . "\n"
        . "  The same options otherwise apply. Examples are:\n"
        . "       w 01-05 00:45:00\n"
        . "       w 2000-01-05 00:45:00\n"
        . "       w 1990  2000-01-05 00:45:00\n"
        . "   etc.\n"
        . "\n"
        . "  Given a WALL timestamp, do two things:\n"
        . "    1. Use Timely to convert to a UTC time;\n"
        . "    2. Use tz to do the same;\n"
        . "  The two can then be compared. As one wall timestamp may have \n"
        . "  two associated UTC values, tz defaults to the later time.\n"
        . "  Timely by default uses the earlier, so an adjustment is made.\n"
        . "\n"
        . "  Errors will result for invalid wall times, for example:\n"
        . "       w 2020-09-27 02:30:00 \n"
        . '';
  }
```

Year and tZ year:

```
elsif( $h eq 'y')
  { $txt = "Check sequential timestamps within a given year, internally. \n"
        . "\n"
        . "  The simplest option is something like   y 2000 \n"
        . "  Year start YYYY-01-01 00:00:00 is converted to a Julian timestamp.\n"
        . "  This is then sequentially increased by increment 'i', each time: \n"
        . "    1. Convert to a Gregorian timestamp for this zone;\n"
```

```
        . "    2. Convert the Gregorian value back to Julian; and \n"
        . "    3. Compare Julian values, issuing a warning if they don't match.\n"
        . "  Dot prints every 50 validations, and newline every $PRETTYDOTS dots.\n"
        . "  \n"
        . "  An internal check with *no* external (e.g. tz) validation. See z too.\n"
        . "\n"
        . "  You can even specify a range of years e.g. y 2000 2010 as well as\n"
        . "  full timestamps along the lines of:\n"
        . "       y 2020-07-01 00:00:00  2021-06-30 23:59:59\n"
        . '';
  }
elsif( $h eq 'z')
  { $txt = "Like 'y', but sequential timestamps are *validated*. \n"
        . "\n"
        . "  As with 'y' you can say e.g.   z 2000   and things like: \n"
        . "      z 2020-07-01 00:00:00  2021-06-30 23:59:59\n"
        . "      z 2020  2022\n"
        . "      z 1900-06-01 00:00:00  1901\n"
        . "      z 1900  1900-01-01 23:59:59\n"
        . "  If just YYYY is specified, default is year start/end \n"
        . "\n"
        . "  The UTC start value is converted to a Julian timestamp.\n"
        . "  This is then sequentially increased by 'i', and each time: \n"
        . "    1. The UTC time derived from the Julian is converted to both a \n"
        . "       Timely wall time and a tz wall time, for this zone. \n"
        . "    2. A warning is issued if they're not the same.\n"
        . "  Print dot every 50 validations, newline every $PRETTYDOTS dots.\n"
        . '';
  }
```

Rather special:

```
elsif( $h eq 'frill')
  { $txt = "                    A few 'frills' and 'factoids': \n"
       . "  ---------------------------------------------------------------------\n"
        . "\n"
        . "  NB: Z and DST offsets apply *up to* the database transition time\n"
        . "  Use  l   to list Leap seconds\n"
        . "  Use  u YYYY-MM-DD hh:mm:ss   to enter UTC timestamp, like g\n"
        . "\n"
        . "       + 1200       Add a number of seconds to menu timestamp\n"
        . "       - 60         Similarly can move earlier, here by 60 s\n"
        . "       + 23:59:59   Can also add hours, minutes and seconds\n"
        . "       + 2.5 days   Add 2.5 days \n"
        . "\n"
        . "  With f for find, add a question mark to print all the matches!\n"
        . "     ... and for an exact match try e.g.   f 'EST' \n"
        . "     ... and you can find by Timely zone ID number too, FWIW.\n"
        . "\n"
        . "  With e.g.  y 2000   any - and + signs signal testing around DST \n"
```

```
          . "  and zone transitions. With small  i  values, these can burgeon! \n"
          . "  Similarly, '>>' signals that in comparing times, the Timely \n"
          . "  value was 'adjusted up' to make it compatible with the tz one.\n"
          . "  (With DST/Z decrements, two UTC times map to one wall time!)\n"
          . "\n"
          . "  The topmost year with t (for Test) is one greater than that with \n"
          . "  w for Wall time because e.g. 12-31 23:59:59 may anticipate UTC \n"
          . "  by several hours in the latter case. \n"
          . '';
    $h = '!';
  }
```

Done:

```
else  # exceptional failure, won't happen without programming error above:
  { print "\nUnknown help option: '$h'\n";
    return(''); # just in case.
  };

  my($M) =
    "\n  ====================-----     HELP ($h)      -----==================="
  . "\n  $txt"
  . "\n  ------------------------------------------------------------------------\n";

  print($M);

  return(''); # success
}
```

### 5.4.5 Clear screen

```
sub Cls #
{
if(Timely::CheckOS() eq 'Windows') # or use $OPERATING_SYSTEM [?]
  { system 'cls';
  } else
  { system 'clear';
  };   # or simply and obscurely:   system $^O eq 'MSWin32' ? 'cls' : 'clear';
}
```

### 5.4.6 Setup menu

Manufacture a text menu. Important arguments are:

**NUMBR** the current GPS timestamp

**$region** the zone.

Other arguments include the sampling interval, the place, and an optional welcome message—a frill. The menu will printed to the console by the calling routine.

```
sub SetMenu #
{ my($NUMBR, $region, $INTERVAL, $placename, $welcome) = @_;

  my($DayNumber) = &JulianDay($NUMBR);

  my($hi) = '     TIMELY      ';
if($welcome)
  {  $hi =  'Welcome to TIMELY';
  };

  my($utc) = Timely::Greg($NUMBR, 0, 0, 1);  # 1=GPS is ON
  my($sec) = &GpsOffset($NUMBR);

 my($wall) = '?';
  my($gYYYY,$gM,$gD,$gh,$gm,$gs, $Z_OFF, $DST, $hog, $deltz, $delds)
                                              # stubs ^       ^
        = Timely::J2G(1, $region, $NUMBR); # Z_OFF,DST=day parts
if($gYYYY)
  { $wall = Timely::PrettyDate(0,$gYYYY,$gM,$gD,$gh,$gm,$gs); # cf. InternalJulian
    #print "Wall in SetMenu is $wall\n";
    my($yours) = Timely::FetchTzDate($NUMBR, $placename, $gYYYY, $gM, $gD,
                                     $gh, $gm, $gs);  #
    #print "FetchTzDate gave $yours\n";
    #
    $yours =~ s/T/ /;
  if($yours eq $utc)
    { $wall .=  $TICK;
    }
  elsif
    ( ($region != Timely::GetUtcCode() ) # don't try with UTC base
     ## && ! $hog   ## <-- a conceit!
    )
    { # $wall .= " (? $yours)";   # " ? $yours"; # temporary.
      $wall .= &Colourise('?', ANSIRED);
    };
  if($hog)
    {   # Esc code will be \033[31
      my($foo) = &Colourise('?', ANSIRED);  # see quotemeta below \Q \E
    if($wall =~ /^(.*\d\d)\Q$foo\E/)
         # if contains red '?' [hmm] AND is groundhog, consolidate to:
      { $wall = $1 . &Colourise('*', ANSIRED); # red star = 1st groundhog!
      } else
      { $wall .= '*';
      };
    };
  };

  # eye candy: extra spaces as length of placename goes from 30 down to 7
  my($filler) = Timely::Padding(' ', (30 - (length $placename))/2 );
```

```
  # here determine DST and Z
  # ...
  my($mydst) = &Tim($DST);
  my($myz) = &Tim($Z_OFF);
```

Noteworthy is the invocation of J2G (), which in turn calls FullGregorian (). J2G () also invokes Anomalous (), which looks up the Z and DST values for the supplied proleptic GPS timestamp in `$NUMBR`. It's important to note that in the interval +DST following a back transition at time T (DST is now cancelled), the wall time will fall back by DST. This obviously only completes once we've reached T+DST. It is thus very valuable to signal back from J2G () if we're in this transition interval. In other words, we need to:

1. Check whether the supplied time is within the last transition + D[3]; and
2. If so, signal this.

The menu itself:

```
  $placename = &Colourise($placename, ANSIBLUE);

   ($region, $placename, $INTERVAL, $utc, $DayNumber, $sec, $wall, $myz, $mydst) =
    &PrettyAll($region, $placename, $INTERVAL, $utc, $DayNumber,
               $sec, $wall, $myz, $mydst);

  my($M) =
    "\n  ========================================================================="
  . "\n  ====================----- $hi -----===================="
  . "\n                $filler ($region : $placename)"
  . "\n                  q=QUIT  h=HELP r=READ  t=TEST  n=Now"
  . "\n  ----------------------------settings-----------------------------"
  . "\n  f name     eg. f Auckl      Find & set region "
  . "\n  g YYYY-MM-DD hh:mm:ss       enter Gregorian wall timestamp (u=UTC)"
  . "\n  j number   eg. g 2415020.5  enter Julian GPS day number"
  . "\n  i interval eg. i $INTERVAL       set year test Interval (seconds)"
  . "\n  -------------------------------tests-------------------------------"
  . "\n  y year     eg. y 2000       test current zone, Year at interval i"
  . "\n  z year     eg. z 2000       tZ timestamp validation at interval i"
  . "\n  a year     eg. a 2000       show transitions Around year"
  . "\n  v     the lot: v all        Validate vs source, try v+1 OR v-1 too"
  . "\n  t MM-DD hh:mm:ss            Test *all* eg. t 12-31 23:59:59"
  . "\n  w MM-DD hh:mm:ss            as for t but Wall time!"
  . "\n  ------------------------------------------------------------------"
  . "\n  UTC : $utc <= GPS $DayNumber delta=$sec s"
  . "\n  Wall: $wall   Z=$myz   DST=$mydst"
  . "\n  =========================================================================\n";
  return($M);
}
```

[3] In the case of D now returning to zero, or, at least a value less than the prior DST, thinking of "super DST" which occurred, for example, in the UK during the Second World War

#### 5.4.6.1 "Tim"

This converts a Julian time fraction (expressed as seconds) to minutes; formerly the submitted values were in fractions of a day.

```
sub Tim #
{ my($t) = @_;

  return($t/60); # just integer seconds.
}
```

Some time offsets (especially older ones) are not limited to integral minute values (cf. Australia/Currie) so we allow two decimal places.

### 5.4.7 Pretty

#### 5.4.7.1 DoubleDigit

(Also present in timely_400.lyx)

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

#### 5.4.7.2 Pretty things

Embolden values that changed: a convenience. Given a fixed number (n=9) of variables, embolden and update them—unless working under Windows.

```
sub PrettyAll  # NB. checks length but not *order* of submitted parameters :(
{ my(@LOTS) = @_;

if($OPERATING_SYSTEM eq 'Windows') # [clumsy]
  { return(@LOTS);   # just return.
  };

  my($N) = scalar @LOTS;
if( $N != scalar @OLDMENUVALUES )
  { die "Missing parameter for PrettyAll_(), n=" . $N ;
    return;
  };
  my($i) = 0;
while($i < $N)
  { $LOTS[$i] = &PrettyBold($LOTS[$i], $i);
    $i ++;
  };
  return(@LOTS);
}
```

The subsidiary emboldening:

```
sub PrettyBold #
{ my($new, $i) = @_;
if( $OLDMENUVALUES[$i] ne $new )  # Global :(
  { $OLDMENUVALUES[$i] = $new; # update
    $new = "\033[1m" . $new . "\033[0m";  # ANSI escape (bold) ... ESC 'reset' code.
  };
  return($new);
}
```

#### 5.4.7.3 Colourise

30–37 = black, red, green, yellow, blue, magenta, cyan, white.

```
sub Colourise #
{ my($txt, $clr) = @_;

  # [might check $clr is valid]

if($OPERATING_SYSTEM ne 'Windows')
  { $txt = "\033[" . $clr . "m" . $txt . "\033[0m";
  };
  return($txt);
}
```

#### 5.4.7.4 Julian Day

Convert seconds to Julian day number: a convenient wrapper. (Is duplicated in *timely_400.lyx*).

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

#### 5.4.7.5 Usage

If you enter a Julian value of e.g. 2415020.5 you should by default obtain the start of the 20th century; note however if you set the zone offset to 60 minutes, then the time in this zone for the same Julian value will be 1 hour later. Zone offsets may be positive or negative, but DST (which has a similar effect) can only be positive.

Setting GPS (t 1) will have the effect of converting e.g. 1900-01-01 00:00:00 not to 2415020.5 but to a value that is nine seconds earlier, as proleptic GPS time is here earlier. Now try the same with eg. 2000.

## 5.5 Kept rules

This section allows us to retain tz rules, check and update/retire them. There are two classes of rule: tz_rules and tz_cutoffs, with corresponding tables. We can thus invoke any of:

**tz_rule_save ()** Save a rule—if present, simply make sure it's active, and return the code (ID), otherwise, create a new one

**tz_rules_inactivate_all ()** Clear the is_active flag to zero for all entries

**tz_cutoff_save ()** Save cutoff rule, as for tz_rule_save (), returning the internal ID

**tz_cutoff_inactivate_all ()** Clear is_active flag to zero for all entries

### 5.5.1 Save tz rule

Save/update an existing rule, stored in the ***f*ehr** database.

```
sub tz_rule_save #
{ my($rule_name, $from_year, $to_year, $in, $on, $at, $dst) = @_;

  my($handDB) = $smalltime_h;

  my($inonat) = "$in $on $at"; # note space separators [explore]

  my($q) = "SELECT tz_rule FROM tz_rules WHERE rule_name = '$rule_name' "
           . "AND from_year = '$from_year' "
           . "AND to_year = '$to_year' "
           . "AND in_on_at = '$inonat' ";
  my($tz_rule) = Timely::GetSQL($handDB, $q, 'find old rule');
if($tz_rule)
  { return($tz_rule); # found
  };

  ## ??????????????????????????/ what if the DST has changed for a rule?!

  # else need to make new entry:
  ($tz_rule) = Timely::FetchKey('tz_rules', 'tz rules');
    # [here might check this succeeded..]
  my($iq) = "INSERT INTO tz_rules "
            . "(tz_rule, rule_name, from_year, to_year, in_on_at, dst) "
            . "VALUES ($tz_rule, '$rule_name', '$from_year', '$to_year', "
            .      "'$inonat', '$dst')";
if( ! Timely::DoSQL($handDB, $iq, 'new tz rule') )
  { die "Failed to insert new tz rule: "
        .  "$rule_name, $from_year, $to_year, $in, $on, $at, $dst";
  };
  return($tz_rule);
}
```

### 5.5.2 inactivate tz rules

```
sub tz_rules_inactivate_all #
{
  my($handDB) = $smalltime_h;

  my($q) = "UPDATE tz_rules SET is_active = 0 WHERE 1";
if(Timely::DoSQL($handDB, $q, "reset all tz rules") < 0)
  { die "Failed to reset tz_rules";
```

```
  };
}
```

### 5.5.3 Save cutoff rule

Apart from the PK tz_cutoff, only the region is an integer.

```
sub tz_cutoff_save #
{ my($stdoff, $rule_name, $region, $until_text) = @_;

  my($handDB) = $smalltime_h;

  # unfortunately:
if(! defined $until_text)
  { $until_text = '';
  };

  my($q) = "SELECT tz_cutoff FROM tz_cutoffs WHERE stdoff='$stdoff' "
             . "AND rule_name = '$rule_name' "
             . "AND region = $region "
             . "AND until_text = '$until_text'";

  my($tz_cutoff) = Timely::GetSQL($handDB, $q, 'find old rule');
if($tz_cutoff)
  { return($tz_cutoff); # found
  };

  # new entry:
  ($tz_cutoff) = Timely::FetchKey('tz_cutoffs', 'tz cutoffs');
    # [here might check this succeeded..]
  my($iq) = "INSERT INTO tz_cutoffs "
            .     " (tz_cutoff, stdoff, rule_name, region, until_text) "
            . "VALUES ($tz_cutoff, '$stdoff', '$rule_name', $region, '$until_text')";
if( ! Timely::DoSQL($handDB, $iq, 'new tz cutoff') )
  { die "Failed to insert new tz cutoff: $stdoff, $rule_name, $region, $until_text";
  };
  return($tz_cutoff);
}
```

### 5.5.4 Inactivate cutoff rules

```
sub tz_cutoff_inactivate_all #
{
  my($handDB) = $smalltime_h;

  my($q) = "UPDATE tz_cutoffs SET is_active = 0 WHERE 1";
if(Timely::DoSQL($handDB, $q, "reset all cutoff rules") < 0)
  { die "Failed to reset tz_cutoffs";
  };

}
```

## 5.6 Populate timely

If we 'unzip' the TZ database, we obtain several files from the data file (e.g. *tzdata2014c.tar.gz*) and from the code file (e.g. *tzcode2014c.tar.gz*). The former refer to:

1. Time zone boundaries;
2. UTC offsets;
3. Daylight saving.

In the relevant data files, the # character is used to signal a comment. The code file allows you to build, install and test the C code on a Linux system; particularly useful is the file *zic.8.txt*. My task is to encode all relevant data in two tables: **timely** (which encodes time zones and daylight saving) and **leapseconds** which simply records transitions of UTC leap seconds.

The tz files can be installed on a Linux system. Note that the data are not authoritative, and somewhat patchy before 1970 — the authors have simply done the best they can. There is a short README (text) and *zone.tab* describes the various zones by numeric code, coordinates, TZ (e.g. Europe/Andorra; Antarctica/McMurdo, Antarctica/Rothera, etc etc ).

### 5.6.1 zo*ne.tab*

Our first task is to allocate unique codes for each of the one or more zones that map to a country.

### 5.6.2 region files

These files are named *africa*, *antarctica*, *asia*, *australasia*, *europe*, *northamerica*, *pacificnew*, and *southamerica*. The pacificnew file is an oddity (see it). The *australasia* file also includes the pacific islands (oceania). NB. As they stand, the rules do not accommodate general rules like e.g. +45 minutes for NZ Chatham Islands, so there are duplicate rules (See the relevant file). More generally, apart from Links, there are # comments, Rules and Zones. Consider e.g. the file *antarctica*.

### 5.6.3 Rules

These have the following properties:

**name** The second entry after "Rule", names the rule;

**from/to** The duration of applicability, by Gregorian year;

- if only one year, the TO field has value "*only*"
- *min*imum (any variant) specifies minimum year — but this variant seems not to be used!
- *max*imum is the maximum year (presumably, "till the topmost year specified")

**type** (ignore, should be "-"); theoretically checks whether the year conforms to a type specification;

**IN** the relevant month. Months can be abbreviated eg Jan, Feb, ...

**ON** the relevant day of that month. This may be a day e.g. 30 (ie. 30th), or:

**lastSun** i.e. "last Sunday of the month", similarly lastMon etc.

**Sun>=n** e.g. "Sun>=8" means the second Sunday of the month

**Sun<=n** last Sunday on or before n (but this seems never to be used!)

**Mon** etc. Any weekday is valid. Sat Sun Mon Tue Wed Thu Fri

**AT** The time on that day, according to the *local* clock (eg. 2 *or* 2:00 *or* 15:00 *or* 1:28:14; 0 is midnight at the start of the day, 24 is midnight at the end of the day) , unless there is the suffix:

**s** local standard time (wall clock time without the DST!)

**g** All of the following {g u z} mean UTC / GMT / nautical time zone Z (the same)

**u**

**z**

**w** The wall clock time, the same as leaving out the suffix

**save** The wall clock offset with the rule in effect (0 for standard time ie no delta; otherwise the daylight saving value, usually 1:00).

**letter/s** This field can take on various values, to specify how the time zone's name changes, e.g. CST vs CDT. These letters are 'aesthetic' in that they specify names but don't condition how the data are translated into the **timely** table.

**-** n/a

**D**

**S**

**W** War time

**P** Peace time !

The rules have sequential entries. As an example:

| # Rule | NAME | FROM | TO | TYPE | IN | ON | AT | SAVE | LETTER/S |
|---|---|---|---|---|---|---|---|---|---|
| Rule | ArgAQ | 1964 | 1966 | - | Mar | 1 | 0:00 | 0 | - |
| Rule | ArgAQ | 1964 | 1966 | - | Oct | 15 | 0:00 | 1:00 | S |
| Rule | ArgAQ | 1967 | only | - | Apr | 2 | 0:00 | 0 | - |
| Rule | ArgAQ | 1967 | 1968 | - | Oct | Sun>=1 | 0:00 | 1:00 | S |
| Rule | ArgAQ | 1968 | 1969 | - | Apr | Sun>=1 | 0:00 | 0 | - |
| Rule | ArgAQ | 1974 | only | - | Jan | 23 | 0:00 | 1:00 | S |
| Rule | ArgAQ | 1974 | only | - | May | 1 | 0:00 | 0 | - |
| Rule | ChileAQ | 1972 | 1986 | - | Mar | Sun>=9 | 3:00u | 0 | - |
| Rule | ChileAQ | 1974 | 1987 | - | Oct | Sun>=9 | 4:00u | 1:00 | S |

### 5.6.4 Zone data

The files contain lines with:

**Zone** A key-word e.g. Australia/Adelaide. The following lines also apply to the same Zone until a new entry is encoutered.

**NAME** the name of the zone

**GMTOFF** the standard GMT offset (from GMT/UTC), can be negative. e.g. -5:50:36 or -6:00

**RULES** The relevant rule (as above), a positive time offset like "1:00" (We *have* set our clocks ahead), or a "-" (no change).

**FORMAT** the abbreviated name, which may contain "%s" to permit substitution of the LETTER/S from above. There are actually 4 possible forms:

**zzz** signals "null";

**LMT** or whatever, an alphabetic string that is *not* "zzz"

**XXX/YYY** a pair of strings, the first for standard time, the second for the DST abbreviation

**X%sY** where "%s" will be replaced by the relevant text from the rule's LETTER column!

**UNTIL** The specified state is in effect *until* the "UNTIL" entry, eg. until 1883 Nov 18 12:09:24 . After this, the state of the *next* line is in force. This is made more difficult as if we're setting the clocks back, there are two possible times — here the first occurrence rules. If there is no UNTIL entry, this means "continue till the present" (and potentially, beyond).

The Zone data look like this:

| # Zone (NAME) | GMTOFF | RULES | FORMAT | [UNTIL] |
|---|---|---|---|---|
| Zone Antarctica/Casey | 0 | - | zzz | 1969 |
| | 8:00 | - | WST | 2009 Oct 18 2:00 |
| | | | | # W (Aus) Standard Time |
| | 11:00 | - | CAST | 2010 Mar 5 2:00 |
| | | | | # Casey Time |
| | 8:00 | - | WST | 2011 Oct 28 2:00 |
| | 11:00 | - | CAST | 2012 Feb 21 17:00u |
| | 8:00 | - | WST | |
| Zone Antarctica/Davis | 0 | - | zzz | 1957 Jan 13 |
| | 7:00 | - | DAVT | 1964 Nov # Davis Time |
| | 0 | - | zzz | 1969 Feb |
| | 7:00 | - | DAVT | 2009 Oct 18 2:00 |
| | 5:00 | - | DAVT | 2010 Mar 10 20:00u |
| ... | | | | |
| Zone Antarctica/Palmer | 0 | - | zzz | 1965 |
| | -4:00 | ArgAQ | AR%sT | 1969 Oct 5 |
| | -3:00 | ArgAQ | AR%sT | 1982 May |
| | -4:00 | ChileAQ | CL%sT | |

#### 5.6.4.1 Notes

From the horse's mouth:

- For areas with more than two types of local time, you may need to use local standard time in the AT field of the earliest transition time's rule ***to ensure that the earliest transition time*** recorded in the compiled file is correct;
- If, for a particular zone, a clock advance caused by the start of daylight saving coincides with and is equal to a clock retreat caused by a change in UT offset, zic produces a single transition to daylight saving at the new UT offset (without any change in wall clock time). To get separate transitions use multiple zone continuation lines specifying transition instants using universal time.

#### 5.6.4.2 Link lines

This starts with "Link" and then has the LINK-FROM and LINK-TO entries. The LINK-FROM entry must be a NAME of a Zone; the LINK-TO entry is simply an alternate name (alias) for that zone.

### 5.6.5 others

The *leapseconds* file simply records the year, month and day of the relevant leap second (all have been +ve so far, but we must accommodate a possible -ve); an alternative source (in a different format) is *leap-seconds.list*. There are some specialised files, including *backward*, which gives older time zone names; and *etcetera*, which contains codes for odd cases like ships at sea. We do not currently accommodate these.

## 5.7 Perl tz extraction

### 5.7.1 Perl overview

Here we convert tz files to table entries. The basic ideas:

1. Read each data file from {*africa*, *antarctica*, *asia*, *australasia*, *europe*, *northamerica*, *southamerica*}. Ignore all comments, then parse each file, line-by-line.
2. Place rules in an associative array, allowing access to rules by name (primarily) and then secondarily by "from/to" entry. Practically, this means an associative array for the rule name, and a sub-array (sequential) for all of the "sub-rules".
3. Work through each Zone entry, parsing each line as a "secondary rule";
4. Resolve Link entries;
5. For each Zone, from 1900 (or earlier) to the current time, establish a unique line for each year, applying all rules, and enter this line in the **daytime** table.
   (a) If the line already exists, check it and report all differences. Have an overwrite/warn rule.

### 5.7.2 Establish zones

The file *zone.tab* contains a list of all TZ zones, mapped to both a country code and a set of coordinates. The format is:

| #code | coordinates | TZ | comments |
|---|---|---|---|
| AD | +4230+00131 | Europe/Andorra | |
| AE | +2518+05518 | Asia/Dubai | |
| AF | +3431+06912 | Asia/Kabul | |
| AG | +1703-06148 | America/Antigua | |
| AI | +1812-06304 | America/Anguilla | |
| AL | +4120+01950 | Europe/Tirane | |
| AM | +4011+04430 | Asia/Yerevan | |
| AO | -0848+01314 | Africa/Luanda | |
| AQ | -7750+16636 | Antarctica/McMurdo | McMurdo, S Pole, Scott (NZ time) |

The coordinates have a sign followed by 4–7 characters:

+/- north/south (first) then east/west.

**4** DDMM

**5** DDDMM

**6** DDMMSS

**7** DDDMMSS

We enter each zone as follows:

1. Read a line, and determine the country code from the two-letter code in the first column;
2. Read the north latitude and east longitude co-ordinates from the second column, and convert to degrees, minutes and seconds; then add up the fractions to get a decimal degree reading; then divide by 9, multiply by 100187903 and convert to an integer for co-ordinate storage (This all to encode offset as an integer in ***f*ehr** internal value);
3. Unless it already exists, create a new **PLACES** entry, named according to the third column.

If `$zonecode` is zero, then we read and re-install the lot; if a zone is supplied, then we currently limit our diagnostics to this zone, and don't write [explore]. On leaving ReadZones (), we return " (an empty string) on success, a string on failure.

```
sub ReadZones #
{ my($zonecode, $zonedesc) = @_;

  my($handDB) = $smalltime_h;

 my($cUPDATE) = 0; # No. of rows updated
  my($cINSERT) = 0; # and inserts.
```

```
  tz_cutoff_inactivate_all();
  tz_rules_inactivate_all();

  my($SingleZone) = ($zonecode != 0);

  my($zonefile) = "zone.tab";
  my($hzone);
  open ($hzone, "$TZDIR/$zonefile")
     or die "\n***ERROR*** Zone file '$TZDIR/$zonefile' not found, $!\n";
  my(@ZDAT) = <$hzone>;
  close($hzone);
  my($zl);
for $zl (@ZDAT)
  { chomp($zl);
  if($zl =~ /^\s*#/ )
    { # print ' ';       # skip comment line
    } else
    { &SetZone($zl, $zonedesc);
    };
  };

# ensure 'UTC' exists:
  my($hack) = 'UT +000000-000000 UTC/UTC UTC';
  &SetZone($hack, $zonedesc);
```

SetZone () calls SaveZone (), which stores the zone identity in %ZONECODES; now we use this to check whether there are widowed zones in the database—active zone entries with no representation in this version of tz. We identify an active zone as a **PLACES** entry with a parent (p_amended) value that refers to a country; all countries have place values in the range of 2–$TOPCOUNTRY (We here exclude the artefact UTC, which has a value of 1).

```
if(! $SingleZone)
  { %REZONE = reverse %ZONECODES; # reverse the hash, NB. Global!
    my($zq) = "SELECT place FROM PLACES "
             . "WHERE p_amended between 2 and $TOPCOUNTRY AND reason > -1 ";
    my(@OLDPLACES) = Timely::SQLManySQL($handDB, $zq, 'get existing, active places');
    print "There are " . (scalar @OLDPLACES) . " old places\n";

    my($pl); # NB. this is an array reference
  for $pl (@OLDPLACES)
    { my($ky) = @$pl[0];
      # print "[$ky]";
    if(! exists $REZONE{$ky}) # inactive
      { Timely::Warn(1, "Retiring unused zone, code $ky\n");
        my($kq) = "UPDATE PLACES SET reason = -2 WHERE place = $ky";
      if(Timely::DoSQL($handDB, $kq, "retiring $ky") < 0)
        { &Aagh("Failed to retire zone code $pl", __LINE__, 0);
```

```
        };   # [more fancy ODBC bound statement would work better, btw]
      };
    };
  };
```

### 5.7.3 Establish leap seconds

The TZ database contains a current list of all leap seconds, so we parse this using the routine from Section 5.7.6.1.

```
my($leapfile) = "$TZDIR/leapseconds";
  print("\n Reading leap seconds from '$leapfile'");
  my($hleap);
  open ($hleap, $leapfile) or die "\n***ERROR*** Could not find $leapfile, $!\n";
  my @LS = <$hleap>;
  close $hleap;

  &ReadLeapSeconds(@LS);
```

### 5.7.4 Read data files

We require the directory $TZDIR containing the specified files. Most important are the Rule and Zone lines within these files. ParseRule (5.9.16) simply adds a line to the relevant rule (establishing it in the associative array %RULES if it doesn't exist); The zone routines StartZone (), MoreZone () and EndZone () progressively add more zone information. Each zone row may reference a rule with potentially many component lines.

```
  my($ZN) = '';       # name of current zone
  my($INZONE) = 0;    # are we in a zone? [clumsy]

  my @FILES = ('africa', 'antarctica', 'asia', 'australasia', 'europe',
               'northamerica', 'southamerica');
      # at present ignore weird : factory, 'backward', 'backzone', 'etcetera'.

foreach my $filename (@FILES)
  { print "\n\n**PARSING MYZONE FILE $filename**";
    my $path = "$TZDIR/$filename";
    open(my $fh, '<', $path)
      or die "\nCould not open '$path' for reading: $!";
    my(@DAT) = <$fh>;
    close($fh);

    my($ln);
  for $ln (@DAT)
    {
    if( $ln =~ /^(.*?)\#/ )
      { $ln = $1;         # remove comment
      };
    if($ln =~ /^\s*$/ )   # empty line
      { # Timely::XPrint(0, ' ');
```

```
      }
    elsif($ln =~ /^Link/ )
      { &MakeLink($ln);
      }
    elsif($ln =~ /^Rule/ )
      { ## Timely::XPrint(0, 'R');
      if($INZONE)
        { &EndZone($ZN);
          $INZONE = 0;   # terminate
        };
        &ParseRule($ln);
      }
    elsif($ln =~ /^Zone/ )
      { ## Timely::XPrint(0, 'Z');
      if($INZONE)
        { &EndZone($ZN);
        };
        $INZONE = 1;
        $ZN = &StartZone($ln, $zonedesc);
      }
    elsif($INZONE)   # should be Zone continuation
      { &MoreZone($ln, $ZN, $SingleZone);
      } else
      { # Timely::XPrint(0,  "\n?=<$ln>\n" );
      };
    };  # end of data for this file

  if($INZONE)
    { &EndZone($ZN);
    };
    $INZONE = 0;   # as new file will follow.
  };
```

### 5.7.5 Create database zone rows

Start with debugging. Obtain the zone array or fail; then iterate through each of the zone entries.

```
  # Timely::XPrint(10, "\n *** CREATING DATABASE ZONE ROWS *** ");
  print "\n Checking database zone rows: n=" . (scalar keys %ZONES);

NAMES: foreach my $nm ( sort keys %ZONES )
  { ## start ZONE loop

    my($NM) = $nm;
    my($region) = $ZONECODES{$NM};  # clumsy but better than hack to %ZONES

  if($SingleZone)  # not 'the lot'
    {
    if($nm ne $zonedesc)
      { next NAMES;
```

```
    };
    print "\nParsing: $nm ($region) $LOWYEAR .. $TOPYEAR\n";
  };

if( $NM eq 'UTC/UTC')         # [explore this special case]
  { next NAMES;
  };

  ## Timely::XPrint(10,  "\n Zone=$nm" );
```

Of note is that the following code:

```
 SELECT north FROM PLACES ...
```

... results in failure of the ODBC connection to provide a large negative integer to Perl (Win32::ODBC seems defective), so we cast the offending item to text (char and not varchar, as the latter cast isn't supported by MySQL). The addition of `AND reason > -1` allows suppression of obsolete places.

```
  my($qz) = "SELECT cast(north as char(13)) FROM PLACES "
            . "where place = $region AND reason > -1";
  my($north) = Timely::GetSQL($handDB, $qz, 'get north/south for zone');
if(! defined $north)
  { Timely::XPrint(0,  "\n*ERROR* Bad zone for $nm($region) <$qz>" );
    next NAMES; # [check this branch]
  };
  my($rA) = $ZONES{$NM};
if( ref($rA) ne 'ARRAY' )
  {
  if(! defined $LINKS{$NM} )
     { &TimeWarn( "No zone definition for '$NM' : " . ref($rA) ,1, $NM, 0);
       # retire this zone in fehr:
       my($qx) = "UPDATE PLACES SET reason = -1 WHERE place = $region";
     if( Timely::DoSQL($handDB, $qx, 'retire region') < 0)
       { &Aagh("Failed to suppress region ($qx)", __LINE__, 0 );
       };
       next NAMES;  # next iteration of foreach loop
     };
    print "\nLink: $NM -->" . $LINKS{$NM};
    $NM = $LINKS{$NM};
    $rA = $ZONES{$NM};  # Fails if double indirection or read single *linked* name!
  if(! defined $rA)
    { return(". I couldn't resolve this!\nRepeat  v  after saying:   f $NM \n");
    };
  };
  my(@A) = @$rA; # array of zone information rows
  my($v);

foreach $v (@A)
  { &DebugZone($v, $SingleZone);
```

```
    };
```

In the above we also accommodate symbolic references to other zones using $LINKS, as described in Section 5.7.8.

#### 5.7.5.1 Span the years

We have a rather strange requirement:

- In the Southern Hemisphere, daylight saving time will normally span the bounds of the year, so we carry the preceding DST over to the next year in `$DSTCARRY`

The important routine here is **ZoneYear** (5.8) — as the name suggests, it determines the entire (primary) database row for a given year and zone.

```
  # Calculate each database row over the full range of years, and write to database
    my($year) = $LOWYEAR;
    my($DSTCARRY) = 0;
    my($LASTTRAN) = 0; # most recent transition
    my($ZONEBASE) = 0; # basis for year start calculation by ZoneYear!
  while($year <= $TOPYEAR+1)  # take one year above!
    { ($ZONEBASE, $DSTCARRY)
         = &ZoneYear($SingleZone, $nm, $year, $ZONEBASE, $north, $DSTCARRY, @A);
      # NB. values in $ZONEBASE, $DSTCARRY are in seconds.
```

If the time-zone offset changed in the previous year, update that year now that we have the new time-zone offset.

In any case, bump the year.

```
      $year ++;  # bump
    };
```

#### 5.7.5.2 Done!

```
  };  ## end of ZONE loop
  return('');
}  ## END OF ReadZones ROUTINE.
```

### 5.7.6 Leap seconds routines

Utility routines that handle leap seconds.

#### 5.7.6.1 Parse tz leap seconds

Confirm that the values in `@LEAPDATA` correspond to those in the tz database. Submit all of the rows. Assumes that `@LEAPDATA` has been populated.

```
sub ReadLeapSeconds #
{ my(@LS) = @_;
  my($leap);
  my($BASELEAP) = -9;  # at baseline GPS is 9s before UTC, UTC generally slows
  my($itop) = scalar @LEAPDATA;
  my($ok) = 1;
```

```
  Timely::Log("\n  Debug LEAP DATA length = $itop");

for $leap (@LS)
  {
  if($leap =~ /^(.*?)#/)
    { $leap = $1;
    };
  if($leap !~ /^\w*$/ ) # if non-empty, not just whitespace
    {
    if($leap !~ /Leap\s+(\d{4})\s(\w{3})\s(\d{2})\s(\d{2}):(\d{2}):(\d{2})\s([-+])\sS/)
                         #YYYY    MMM      DD        hh       mm     ss      +   S
      { die "\n***ERROR*** Unknown leap line: <$leap>\n";
        return;
      };
      $itop --; # move to corresponding data row;
      my($d) = $LEAPDATA[$itop];
      my(@D) = @$d; # dereference
      my($utc) = $D[1]/1000000.0;  # seconds
      my($off) = $D[2];
      my($YYYY, $MMM, $DD, $hh, $mm, $ss, $updn) = ($1, $2, $3, $4, $5, $6, $7);
    if($ss != 60)
      { die "\n***ERROR*** Odd leap line: $YYYY $MMM $DD $hh $mm *$ss* $updn";
        return;
      };
    if($updn ne '+')
      { $BASELEAP --;
        die "\n***ERROR*** I can't handle '-' value in leap line";  # [explore, fix]
        return; # need to check handling before use, if this ever happens.
      } else
      { $BASELEAP ++; # Applies to times between $bigJ and the next cut.
      };
    if($BASELEAP != $off)
      { die "\n***ERROR*** mismatched leap seconds: $BASELEAP/$off";
        return;
      };
      my($Mon) = &FetchMonth($MMM);  # e.g. Jan->1
      my($bigJ) = Timely::ToJulian($YYYY, $Mon, $DD, $hh, $mm, $ss, 0, 0, 0);
                # NOT GpsJulian ; and potential 60s value in $ss
    if( abs($bigJ - $utc) > 0.01 ) # if difference is more than 10 ms:
      { my($COOKED) = Timely::Greg($utc, 0, 0, 0); # utc is in seconds [explore]
        Timely::XPrint(0, "\n*ERROR* bad leap timestamp: $bigJ, "
                  . "$utc ($YYYY-$Mon-$DD $hh:$mm:$ss | $COOKED)" );
        $ok = 0;
      };
    };
  };
if(! $ok)
  { die "\n***ERROR*** Error in leap seconds";
  };
}
```

In the above we add $delta to the timestamp because we always store GPS times, not UTC!

#### 5.7.6.2 Populate leap data

Conversely, populate the ***f*ehr** leapseconds table.

```
sub PopulateLeapSeconds #
{ my($TZDIR) = @_;

  my($handDB) = $smalltime_h;  # CLUMSY

  my($leapfile) = "$TZDIR/leapseconds";
  print("\n Populating leap seconds from '$leapfile'");
  my($hleap);
  open ($hleap, $leapfile) or die "\n***ERROR*** No leap seconds source: $leapfile, $!\
  my @LS = <$hleap>;
  close($hleap);

  my($leap);
  my($BASELEAP) = -9;  # Baseline GPS time is 9s before UTC, UTC generally slows
  my($leapsecond) = 0;

for $leap (@LS)
  {
  if($leap =~ /^(.*?)#/)
    { $leap = $1;
    };
  if($leap !~ /^\w*$/ ) # if non-empty, not just whitespace
    {
    if($leap !~ /Leap\s+(\d{4})\s(\w{3})\s(\d{2})\s(\d{2}):(\d{2}):(\d{2})\s([-+])\sS/)
                        #YYYY    MMM      DD        hh       mm     ss      +   S
      { die "\n***ERROR*** Unknown leap line: <$leap>\n";
        return;
      };
      my($YYYY, $MMM, $DD, $hh, $mm, $ss, $updn) = ($1, $2, $3, $4, $5, $6, $7);
    if($ss != 60)
      { die "\n***ERROR*** Unusual leap entry: $YYYY $MMM $DD $hh $mm *$ss* $updn";
        return;
      };
    if($updn ne '+')
      { $BASELEAP --;
      } else
      { $BASELEAP ++; # This applies to times between $bigJ and the next cut.
      };
      my($Mon) = &FetchMonth($MMM);  # e.g. Jan->1
      my($bigJ) = Timely::ToJulian($YYYY, $Mon, $DD, $hh, $mm, $ss, 0, 0, 0); # NOT Gps
               # also take note of the 60s value in $ss. This could be 61 !
      my($gps)= $bigJ+$BASELEAP;     # to get gps, add offset to UTC value
      my($q) = "INSERT INTO leapseconds (leapsecond, gpstime, utctime, toffset)"
     . " VALUES ($leapsecond, $gps*1000000, $bigJ*1000000, $BASELEAP)";
```

```
      # multiply seconds by 10^6 to get stored microseconds.
    if( Timely::DoSQL($handDB, $q, 'insert leapsecond row entry') < 0)
      { &Aagh("Failed to insert leapsecond row ($q)", __LINE__, 0 );
        return;
      };
        # [ideally check entry not already present, and check for success[explore]]
      $leapsecond ++;
    };
  };
}
```

#### 5.7.6.3 GpsOffset

Given a Julian day as seconds, return the GPS offset in *seconds*. This is the number of seconds that GPS time leads UTC for the given time `$J`, which is assumed to be a GPS time, not a UTC time.

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

  my($Ltop) = scalar @LEAPDATA;
  my($i) = 0;
while($i < $Ltop)
  { my($d) = $LEAPDATA[$i];
    my(@D) = @$d; # dereference
  if( $J >= ($D[0]/1000000.0) )  # note 0 index [check inequality for consistency ?]
    { return($D[2]);     # offset in seconds
    };
    $i ++;
  };
  return(-9);  # earliest value is a lag.
}
```

### 5.7.7 Major zone routines

These handle zone data encoded in lines of the TZ database, outlined at the start of Chapter 5.2.

#### 5.7.7.1 Set a zone

SetZone () is invoked when we first encounter a zone name within ReadZone (5.7.2). Given basic data, establish a zone. The principal subroutine here is SaveZone (5.7.7.4). Only later, when more specific zone information becomes available do we invoke StartZone () etc — these routines merely modify the zone information in the associative array of zones, `%ZONES`.

If a zone description is supplied in `$zonedesc`, then we are only interested in that zone, and ignore others.

```
sub SetZone #
{ my($zl, $zonedesc) = @_;
```

```
  my(@ZD) = split /\s+/, $zl;
  my($cc) = $ZD[0];
  my($rawcoords) = $ZD[1];
  my($zonename) = $ZD[2];

if(length $zonedesc > 0)
  {
  if($zonedesc ne $zonename) # if single zone and doesn't match:
    { return(''); # ignore
    };
    print " [Setting zone $zonedesc: '$zl'] ";
  };

  # Timely::Log("\nSetting zone: <$zl>=<$cc><$rawcoords><$zonename>");
if(length $cc != 2)
  { die "\n***ERROR*** Bad 2-char country code: $cc\n";
    return;
  };
if($rawcoords !~ /([+-])(\d+)([+-])(\d+)/ )
  { die "\n***ERROR*** Bad coordinates for $cc: $rawcoords";
  };
  my($sN)   = $1;    # sign
  my($rawN) = $2;
  my($sE)   = $3;
  my($rawE) = $4;
  my($North) = &NEncode($sN, $rawN);
  my($East) = &NEncode($sE, $rawE);
  my($ZID) = &FetchCountryCode($cc);
if($ZID < 0)
  { die "\n***ERROR*** Country code not found: $cc\n";
  };
  &SaveZone($zonename, $ZID, $North, $East); # $ZID is country
  return($zonename);
}
```

The returned value is currently unused.

#### 5.7.7.2 Encode degrees to integer

This conforms to my 'standard' geospatial integer encoding in ***f*ehr**.

```
sub NEncode #
{ my($sgn, $vlu) = @_;
  my($vlen) = length $vlu;
  # Perl 5.24 seems to have issues with $1 $2 $3 in the following ??
  my($deg, $min, $sec);

if($vlen == 4)
  { $vlu =~ /(\d{2})(\d{2})(.*)/;
   # print "(Debug $1:$2:$3)";
```

```
    ($deg,$min,$sec) = ($1,$2,$3);
  }
elsif($vlen == 5)
  { $vlu =~ /(\d{2})(\d{3})(.*)/;
   # print "(Debug $1:$2:$3)";
    ($deg,$min,$sec) = ($1,$2,$3);
  }
elsif($vlen == 6)
  { $vlu =~ /(\d{2})(\d{2})(\d{2})/;
   # print "(Debug $1:$2:$3)";
    ($deg,$min,$sec) = ($1,$2,$3);
  }
elsif($vlen == 7)
  { $vlu =~ /(\d{3})(\d{2})(\d{2})/;
   # print "(Debug $1:$2:$3)";
    ($deg,$min,$sec) = ($1,$2,$3);
  } else
  { die "\n***ERROR*** Bad degree code: $sgn $vlu";
          return;
  };
 # print "(!!Debug $1:$2:$3!!)";
 # my($deg) = $1;
 # my($min) = $2;
 # my($sec) = $3;
if(length $sec == 0)  # ?? can it be undefined
  { $sec = 0;
  };
   # Timely::XPrint(0, "<Encoding '$vlu'($vlen) '$deg:$min:$sec'>");
  my($enc) = 100187903*($deg + $min/60 + $sec/3600)/9;
if($sgn eq '-')
  { $enc = -($enc);                  # use sign, ignore '+' value [? check]
  };
  my($ienc) = sprintf "%.0f", $enc;  # round half to even
  return($ienc);                     # integer value
}
```

#### 5.7.7.3 Fetch country code

Obtain database code for country, given its two-letter country code.

```
sub FetchCountryCode #
{ my($cc) = @_;

  my($handDB) = $smalltime_h;

  my($q) = "SELECT country FROM countrycodes WHERE ccode='$cc'";
  my($DBcode) = Timely::GetSQL($handDB, $q, 'get country code');
if(length $DBcode == 0)
  { return(-1);             # invalid
  };
  return($DBcode);
```

```
}
```

#### 5.7.7.4 Save zone details

First check if zone is present in database. If not, insert it into the **PLACES** table. We use several globals (`$USERID`, $TZDATABASEID). The constraint `reason > -1` will suppress obsolete locations like America/Montreal, which was incorrectly specified in early versions of tz as an individual timezone different from others in Quebec but here this is a two-edged sword—how do we identify and handle existing (but retired) places? One answer is to select the reason instead of saying `AND reason > -1`, and then *not* add (and save) this zone if the place is retired.

```
sub SaveZone #
{ my($zonename, $COUNTRY, $North, $East) = @_;

  my($handDB) = $smalltime_h;

  my($q) = "SELECT place, reason FROM PLACES "
    . "WHERE p_amended = $COUNTRY AND description = '$zonename'";
    # might get away with just description but rather check country too.
  my($zcode, $reason) = Timely::GetSQL($handDB, $q, 'find zone');

if( (defined $zcode)
  &&(length $zcode > 0)
  )
  { Timely::Log("\n  Zone ($COUNTRY:$zonename) found='$zcode'");
  if($reason < 0) # [this option may need checking]
    { Timely::XPrint(0, "\n\n*Warning* : zone is retired: '$zonename'\n");
      return(0);
    };

    $ZONES{$zonename} = 1; # for later
    $ZONECODES{$zonename} = $zcode;
    return(0);  # return value is unused at present
  };

  my($newzone) = Timely::FetchKey('PLACES', 'new zone code');
  Timely::Log("\nInserting new zone: $newzone=$COUNTRY/$zonename, N=$North E=$East");
  my($qw) = "INSERT INTO PLACES "
   . "(place, east, north, description, p_amended, t_amended, reason, "
   .    "amender, src, chk)"
   . " VALUES "
   . "($newzone, $East, $North, '$zonename', $COUNTRY, 0, 0, "
   .    "$USERID, $TZDATABASEID, 0)"; # no chk yet.
  my($ok) = Timely::DoSQL($handDB, $qw, 'make new zone');
if($ok != 1)
  { &Aagh( "\nFailed to insert zone '$zonename' id=$newzone; $!\n", __LINE__, 0);
    return(0);
  };
  $ZONES{$zonename} = 1;   # stub for later
```

```
  $ZONECODES{$zonename} = $newzone;
  return(1); # new zone created
}
```

#### 5.7.7.5 Debug a zone

Explore the basic structure of the zone provided, and write to log.

```
sub DebugZone #
{ my($v, $SingleZone) = @_;
  my(@V) = @$v;
  my($GMTOFF) = $V[0]; # Z in seconds
  my($RULE)  = $V[1];
    my($YYYY)  = $V[2];
    my($MM)    = $V[3];
    my($DD)    = $V[4];
    my($hh)    = $V[5];
    my($mm)    = $V[6];
    my($FORMAT)= $V[7];
    my($J)     = $V[8]; # Julian seconds
  my($UNTIL) = "$YYYY $MM $DD $hh $mm";
  my($RU);

  my($err) = 0;

if($RULE eq '-')
  { $RU = '<>';
  }
elsif(defined $RULES{$RULE})
  {  $RU = '@' . $RULE;
  }
elsif( $RULE =~ /^\d+:\d+/ )
  { $RU = "off=$RULE";
  } else
  { $RU = " -?- ($RULE) ";
    $err = 1;
  };

if($SingleZone)
  { print "zone rule: $GMTOFF $RU: $UNTIL $FORMAT $J\n";
  };

  Timely::Log( "\n   $FORMAT '$GMTOFF' $RU --> '$UNTIL'/$J" );
    # cf. below: my(@V) = ($GMTOFF, $RULES, $UNTIL);
}
```

### 5.7.8 Links

The format is along the lines of:

Link Europe/London Europe/Jersey

This maps Europe/Jersey to the more widely used rules for Europe/London. We create a special associative table %LINKS that allows link look-up:

```
sub MakeLink #
{ my($rw) = @_;
if( $rw !~ /Link\s(\w+\/\w+)\s(\w+\/\w+)/ )
  { die "\n***ERROR*** Bad link <$rw>";
    return;
  };
  $LINKS{$2} = $1;
  Timely::Log( "( $2-->$1 )" );
}
```

This is used in Section 5.7.5.

## 5.8 Zone data for one year

Process zone data for a single year. As we work through, Store (5.8.2) is invoked to temporarily store an entry in @STORAGE; finally, FinalStore (5.8.2.1) works through this array and writes relevant entries to the database table **timely**.

The arguments are:

**SingleZone** Only true (1) if we're dealing with just a single zone

**nm** The $nm variable, the zone name, is just for decoration (comments)

**year** The year being processed

**ZONEBASE** The Z offset for date calculation

**north** A positive or negative value—a positive number if we're north of the equator

**DSTCARRY** carried over daylight saving (seconds)

**@A** An array of all the data for this zone, obtained from %ZONES.

$DSTCARRY is daylight saving time carried over to "the time of this rule". Each such line in @A is pertinent to a series of years up to the transition time specified within this line; any zone line may also specify a "rule" that contains multiple sub-rules, each with applicability to a range of years. The first zone line applies until the cutoff date specified in that line ($Jtrans); if we next encounter a year after this cutoff, we apply the subsequent rule to that year provided the year ($year) is *between* the cutoffs for the two rules. The value in $ZONEBASE is an offset used as a basis for date calculation, a fraction of one day expressed in seconds. It's important to note that $DSTCARRY is copied into $RUNNING_DST, which may be modified.

ZoneYear () **returns** two values, $CURRENT_ZO, $RUNNING_DST: we must maintain the current zone offset and current DST value for the next year.

In the following, take note of the value in $Jraw and its important derivative—the widely-used transition time $Jtrans. This is the Julian equivalent of the transition time for a rule, but needs to be modified by the current DST and zone offset.

```
sub ZoneYear #
{ my($SingleZone, $nm, $year, $ZONEBASE, $north, $DSTCARRY, @A) = @_;

  Timely::Log( "\n\n** Processing $nm $year (z=" . &Tim($ZONEBASE) . '|d='
        . &Tim($DSTCARRY) . "):" );  # nasty
    my($LASTZO) = $ZONEBASE;

  my($LASTTRAN) = 0;
if($SingleZone)
  { print "\n$year: ";
  };

  # 'start/end of the year' won't accord with UTC start unless zone offset=0. Adjust:
  my($YearBot) = Timely::GpsJulian($year,  1,  1,  0,  0,  0, 0, $ZONEBASE, $DSTCARRY);
  my($YearTop) = Timely::GpsJulian($year, 12, 31, 23, 59, 59, 0, $ZONEBASE, $DSTCARRY);
             #                              ^ 1 second taken off "just in case"
             # BUT NOTE that DST may well change for the latter, as may the zone!
  &StartStorage($nm, $year);
   # NB final 2 values submitted to GpsJulian_() are seconds
  ##&Debug(0, ' from ' . &Dat($YearBot) . ' to ' . &Dat($YearTop) );
```

In the above, if we are to be pedantic, we must eventually amend `$YearTop`, but this has no practical effect. Initialise:

```
  my($FORMAT);                # although retrieved, this variable is not used
  my($Jtrans);                  # transition (cutoff) time as Julian
  my($LAST_ZO) = 0;           # [???]
  my($CURRENT_ZO) = $ZONEBASE; # [horrendous, redundant mess]
  my($TOPPED) = 0;            # =1 means "value in $CURRENT_ZO is now final"
```

### 5.8.1 Process each zone rule

Check multiple zone rules in @A. It's important that we work through @A in the sequence in which the zone rules are presented in the TZ database, from early to late.

```
  my($v);
  my($DONE) = 0;
  my($PENDING_START) = 0;
  my($BETWEEN) = 0;
  my($RUNNING_DST) = $DSTCARRY;  # [another ugly monster]
  #&Debug(0, "\nrunDST1=" . &Tim($RUNNING_DST) );
  my($zod);
```

#### 5.8.1.1 Main Zone loop

As noted above the array @A was supplied as the final argument of ZoneYear ().
We work through each row!

```
NEXTZONE: foreach $v (@A)  # Usually ONE zone specification applies to a year
                   #   but tricky with transitions: the specification (e.g. rule)
                   #   for the NEXT year applies after the transition!
```

```
{ my(@V) = @$v;   #  Each 'rule' in turn contains sub-rules, $v a reference
  my($GMTOFF)= $V[0];  # the zone offset applies UP TO the transition, in seconds
  my($RULE)  = $V[1];
  my($YYYY)  = $V[2];  # the year alone is supplied to ApplyRule

  my($MM)    = $V[3];    # These values all redundant but debugging...
  my($DD)    = $V[4];    #
  my($h)     = $V[5];    #
  my($m)     = $V[6];    # # end redundant.

  $FORMAT    = $V[7]; # the standard 8 variables
  my($Jraw)  = $V[8]; # Julian day for the above YYYY-MM-DD etc.
  my($sufx)  = $V[9]; # suffix for this date, added belatedly [D'Oh]
 my($tz_cutoff) = $V[10]; # added for source debugging
  $zod = 1;
  $Jtrans = &FixBySuffix($sufx, $Jraw, $GMTOFF, $RUNNING_DST);
  $LASTZO = $GMTOFF;   # retain zone offset used to determine Jtrans
    # As per the suffix, convert time based on current zone offset and
    #    on current daylight saving ($dst)
    # NB. Latter more complex based as may be subsequent
    #    DST transitions in the rules themselves.
    # $DSTCARRY may be adjusted below, for the next round of NEXTZONE.
  $Jtrans = Timely::ApplyGps($Jtrans); # adjust to GPS time!
  ($CURRENT_ZO, $TOPPED) =
       &TryZone($GMTOFF, $Jtrans, $YearBot, $YearTop, $CURRENT_ZO, $TOPPED);
```

#### 5.8.1.2 Rule evaluation

Our evaluation process is as follows:

1. Zone rules only apply until the specified cutoff timestamp. If the bottom of the year under evaluation is above the cutoff timestamp in Jtrans, the rule can't apply, so we skip it;
2. Otherwise, a '-' rule forces us to alter the transition point (starting point for the next DST offset, we turn off DST up to the transition) to the cutoff timestamp, provided this timestamp is within the current year!
3. Otherwise a rule like '1:00' forces a new DST value that always applies from the preceding transition point until the current cutoff timestamp;
4. Otherwise apply the named rule. This itself is a bit tricky, as a named rule that "doesn't apply" can be followed (on rare occasions) by another zone rule. See for example America/Nipigon 1940.

In the following, it's important not to apply a rule after it's expired. This is more tricky than it seems. Many zone rules simply specify "up to (e.g.) 1900", so we should not apply the rule if we're in 1900, i.e. the bottom of the year is above the first day of 1900. It's important to realise that the

zone offset that applies up till this point is *not* the preceding offset, but `$GMTOFF` (See the construction of `$Jtrans` above).

```
    my($oktm);
  if($YearBot > $Jtrans)    # NOT >= cf Asia/Khandyga 2004
    { #&Debug(0, "skip ROW, " . &Dat($YearBot) . " > " . &Dat($Jtrans)  );
    if($SingleZone)
      { print '^ ';
      };
      next NEXTZONE;
    }
```

#### 5.8.1.3 Dash rule

This applies OFF between the preceding transition (`$LASTTRAN`) and the current cutoff time (in `$Jtrans`), but only if we've entered with DST currently on; there is no value in recording an "end" if DST is already off! Note the storage of `$LASTTRAN`: MUST NOT allow subsequent rules to apply retroactively before this timestamp, for example as incorrectly occurred with Asia/Gaza 2008 around 08-31.

```
  elsif($RULE eq '-')
    { $RUNNING_DST = 0;   # applies until the transition, whenever it is
    if($SingleZone)
      { print '- ';
      };
    if($Jtrans < $YearTop)
      { $Jtrans = Timely::ApplyGps( &FixBySuffix($sufx, $Jraw, $GMTOFF, $RUNNING_DST) )
           # DST changed, but retroactive
      if( &Store($LASTTRAN, $Jtrans, $RUNNING_DST, $CURRENT_ZO, $sufx,
                 DASHRULE, 0, $SingleZone) )
        { #&Debug(0, "\nrunDST2=0" );
          $LASTTRAN = $Jtrans;
        };
      } else       # if at or after year top
      { last NEXTZONE;
      };
    }
```

#### 5.8.1.4 Fixed DST rule

Here, the daylight saving value is bizarrely specified as a "rule", applying retroactively—in contrast, 'Rule' rules apply DST prospectively.

```
  elsif($RULE =~ /\d+:\d+/)   # value applies UP TO $Jtrans
    { $RUNNING_DST = &FixTime($RULE);   # new DST offset in seconds
      $Jtrans = Timely::ApplyGps( &FixBySuffix($sufx, $Jraw, $GMTOFF, $RUNNING_DST) );
         # DST changed, but retroactive [check this, it works]

    if($SingleZone)
      { print ': ';
      };
```

```
    if($Jtrans < $YearTop)
      {
      if( &Store($LASTTRAN, $Jtrans, $RUNNING_DST, $CURRENT_ZO, $sufx,
                 FIXEDDST, 0, $SingleZone) )
        { #&Debug(0, "\nrunDST3=0" );
          $LASTTRAN = $Jtrans;
        };
      } else       # if at or after year top
      { last NEXTZONE;
      };
    }
```

#### 5.8.1.5 Rule "just applies"

Cut is above top of year (Next rule applies retrospectively!!)

```
  elsif($Jtrans > $YearTop) # if rule "just applies":
    {
    if($SingleZone)
      { print '> ';
      };
      ($oktm, $RUNNING_DST)
         = &ApplyRule($nm, $year, $RULE, $RUNNING_DST, 0, $LASTTRAN,
                      $CURRENT_ZO, $LASTZO, $SingleZone);
          # no transition
       #&Debug(0, "\nrunDST5=" . &Tim($RUNNING_DST) );
       last NEXTZONE;
    }
```

#### 5.8.1.6 A transition is present

This transition is between year start and year end. ApplyRule () may well invoke Store () successfully, in which case a transition time is returned and put in `$oktm`. However, if no store occurred, or the storage was unsuccessful, then $oktm is zero.

The test `if($Jtrans > $oktm)` is somewhat mysterious. This will clearly apply if `$oktm` is zero; it will also apply if the current transition time (for this transition) is above the time specified *up to which the most recent rule stored within ApplyRule ()* applies. For example, take the Asia/Gaza rule:

```
2:00 Zion EET/EEST 1948 May 15
```

The Zion rules of apparent relevance are:

```
Rule Zion 1948 only - May 23 0:00 2:00 DD
Rule Zion 1948 only - Sep 1 0:00 1:00 D
Rule Zion 1948 1949 - Nov 1 2:00 0 S
```

None of these applies, but we still wish to signal the 'transition'—admittedly somewhat meaningless. More substantial is e.g. America/Araguaina 1990:

```
-3:00 Brazil -03/-02 1990 Sep 17
-3:00 - -03 1995 Sep 14
```

The rules:

```
Rule Brazil 1989 only - Oct 15 0:00 1:00 -
Rule Brazil 1990 only - Feb 11 0:00 0 -
Rule Brazil 1990 only - Oct 21 0:00 1:00 -
```

ApplyRule () has stored the Feb 11 transition to DST=0; we must still store the September 17 transition that encompasses the DST that applies till then!

```
else                  # more interesting, as CURRENT transition is present:
  {
  if($SingleZone)
    { print '? ';
    };
    ($oktm, $RUNNING_DST) # [ *** EXPLORE AND FIX THIS NASTY HACK *** ]
       = &ApplyRule($nm, $year, $RULE, $RUNNING_DST, $Jtrans, $LASTTRAN,
                    $CURRENT_ZO, $LASTZO, $SingleZone);
     #&Debug(0, "\nrunDST6=" . &Tim($RUNNING_DST) );
     # KNOW  $Jtrans < $YearTop
   # a catch if rule already set AFTER $Jtrans:
  if($Jtrans > $oktm)
    { #&Debug(0, ' ===unfulfilled=== ' );
      $Jtrans = Timely::ApplyGps( &FixBySuffix($sufx, $Jraw,
                           $GMTOFF, $RUNNING_DST) ); # DST changed?
      ##
    if( &Store($LASTTRAN, $Jtrans, $RUNNING_DST, $CURRENT_ZO, $sufx,
               FIXEDDZ, 0, $SingleZone) )
      { $LASTTRAN = $Jtrans;
      } else
      { # [debug statement will appear within &Store]
      };
    } else
    { # #&Debug(0, "\n --********skipped********-- as " . &Tim($Jtrans)
      #       . ' < ' . &Tim($oktm) );
      $LASTTRAN = $oktm;
    };
  };
```

Continue:

```
};   # End of NEXTZONE loop.
```

We now store each rule in the database. There's a catch, as the DST value used to establish `$YearTop` may have been inaccurate, so I adjust this appropriately. I also need to pass on the last transaction timestamp (`$LASTTRAN`) as explained in FinalStore (5.8.2.1).

```
  $YearTop = Timely::GpsJulian($year, 12, 31, 23, 59, 59, 0, $CURRENT_ZO, $RUNNING_DST)
  &FinalStore($nm, $year, $YearBot, $YearTop, $RUNNING_DST,
              $CURRENT_ZO, $LASTTRAN, $SingleZone);
```

That's it:

```
  return($CURRENT_ZO, $RUNNING_DST);
}
```

#### 5.8.1.7 Start storage

Because of the multiple entries in this table, we will have to clumsily delete all **timely** entries for this zone and year before we create new entries.

```
sub StartStorage
{ my($zone, $year);
    ($zone, $year) =@_;

  @STORAGE = ();  # clear storage
}
```

### 5.8.2 Push to store

Arguments are:

**$LASTTRAN** The most recent transaction Julian timestamp in seconds

**$tran** The current transition time (also Julian seconds).

**$dst** DST prevailing at the transition, in seconds

**$zoff** Z in seconds, the zone offset at the transition

**$mod** Date modifier (tz suffix code e.g. 's')

**$type** Type of change (see below)

**$term** 0 unless the invoker is FinalStore () — then it's 1.

**$SingleZone**

Put a rule in the global `@STORAGE`. As a precautionary measure I supply a `$LASTTRAN` value as the first argument. If the transition timestamp is before this, a warning is issued, and nothing is stored! The return value is the transition time, or 0 on failure. The `$type` is the type of change (an added check); `$mod` is the modifier for the date. The type constant specifies the context in which Store () was invoked:

**DASHRULE** A 'dash' rule;

**FIXEDDST** A 'fixed DST' rule;

**FIXEDDZ** A current, fixed transition;

**DSTTRAN** A DST transition;

**FINALRULE** A mandatory "final store", invoked by FinalStore (5.8.2.1) to ensure there's always a year-end entry, which minimises searching.

A concluding issue concerns the case where two transitions occur at the same time—for example in Asia/Aqtau we have the zone rules:

```
5:00 RussiaAsia +05/+06 1994 Sep 25 2:00s
4:00 RussiaAsia +04/+05 2004 Oct 31 2:00s
```

... as well as:

```
Rule RussiaAsia 1984 1995 - Sep lastSun 2:00s 0 -
```

In other words the zone offset decreases by an hour *and* DST turns off at 1994-09-25 02:00 (wall time ignoring DST). If we process these separately, there's also the danger that we miscalculate the two transition times as different, but the main issue is that we need to *combine* the DST+Zone changes. These issues are all handled by FinalStore (5.8.2.1).

```
sub Store #
{ my($LASTTRAN, $tran, $dst, $zoff, $mod, $type, $term, $SingleZone);
    ($LASTTRAN, $tran, $dst, $zoff, $mod, $type, $term, $SingleZone) =@_;

  #&Debug(0, "\n    ==>STORE " . &Dat($tran) . ' d=' . &Tim($dst) . ' z='
  #      . &Tim($zoff) . " $mod/$type" );

if($tran < $LASTTRAN)
  { Timely::Warn(0, "Antecedent rule compensation: " . &JulianDay($tran)
           . " " . &JulianDay($LASTTRAN) . " " . &Dat($tran)
           . ' < ' . &Dat($LASTTRAN) );
    return(0); # no!
  };

  ## We now store *seconds* in first three values in @STORAGE rows:
  my @INS = ($tran, $dst, $zoff, $mod, $type, $term);  # array to insert
  my($off) = scalar @STORAGE; # get size of @STORAGE
  my($more) = 1;
while(  $more
      &&($off >= 1)
     )
  { $off --;
    my($v) = $STORAGE[$off]; # get top
    my($t, $d, $z) = @$v;
  if($tran >= $t) # IF the same (=) then insert above current
    { $more = 0; # terminate, the usual behaviour
      $off ++;   # will insert ABOVE current element
    };
  };
  splice @STORAGE, $off, 0, \@INS;  # if $off is zero, inserts at start
  return($tran);
}
```

#### 5.8.2.1 Reconcile & write

The final transaction must always be after the most recent transaction, so we check for this using `$LASTTRAN`. If we don't do this, then a transition on 31 December will fail to take effect. An example is America/Matamoros 1921. There are two further issues:

1. It is possible that a timestamp may have been miscalculated, for example if we altered the zone offset and then changed DST, the adjusted transition for the DST will be out by the zone change. We need to fix this and *then* check for identical transition timestamps. Our algorithm:
   (a) Work through, and where zone changes, record this, together with the transition time;
   (b) If subsequent transition + transition change is equal to transition time, then replace transition time for that row. An example occurs for Asia/Aqtau at 1994-09-25 02:00:00s.
2. If two entries in `@STORAGE` (a stack, in order) have the same timestamp, this *may* just be redundant, but we must ensure that if they represent two different transactions (e.g. change DST, change Zone offset) these are combined into one entry in **timely**;

The third to seventh arguments (`$YearBot` – `$LASTTRAN`) are all expressed in seconds, not Julian day numbers!

```
sub FinalStore #
{ my($zone, $year, $YearBot, $YearTop, $CURRENT_DST, $CURRENT_ZO,
     $LASTTRAN, $SingleZone);
    ($zone, $year, $YearBot, $YearTop, $CURRENT_DST, $CURRENT_ZO,
     $LASTTRAN, $SingleZone) =@_;

  my($handDB) = $smalltime_h;

if($SingleZone)
  { print "\n  (.)";
  };

if($LASTTRAN > $YearTop)
  { $YearTop = $LASTTRAN + 1; # +1 second, always above most recent 'true' row
  };

  &Store($LASTTRAN, $YearTop, $CURRENT_DST, $CURRENT_ZO, '', FINALRULE,
         1, $SingleZone);  # cap entries, discard return value
  # the above will succeed, NB. assures >= one entry per zone per year!

if($SingleZone)
  { print "\n[n = " . (scalar @STORAGE) . '] ';
  };
```

Debugging:

```
  my($region) = $ZONECODES{$zone};

if(! $SingleZone)  # do NOT alter SQL if single zone:
  { my($qdel) = "DELETE FROM timely WHERE region = $region AND year = $year";
  if( Timely::DoSQL($handDB, $qdel, 'delete old entries') < 0)
    { &Aagh("Failed to delete old entries ($qdel)", __LINE__, 0);
      return;
    };
  };
```

For each storage item, perform some checks. The “Redundant tz entry” message will be triggered if two identical rules exist, for example the specification for the zone America/Argentina/Buenos_Aires includes:

```
-4:00 - -04 1930 Dec
```
and
```
-4:00 Arg -04/-03 1969 Oct 5
```

The latter specifies the rules for Arg that include:

```
Rule Arg 1930 only - Dec 1 0:00 1:00 -
```

This rule translates to “daylight saving terminates on 1 Dec at 0:00 wall time, in other words, GMT-4; but the first of the above rules “applies” GMT-4 (with no other specifications) until “December 1930” without any other specification. My code currently translates this last to the same timestamp, and registers a ‘collision’.

```
  my($ins);
  my($kept_tran, $kept_dst, $kept_zoff)=(0,0,0);
foreach $ins (@STORAGE)
  { my($tran, $dst, $zoff, $mod, $type, $IGNORED) = @$ins;   # retrieve stored values
    # Retrieved values for $tran, $dst, $zoff are in *seconds*

    # Sort out the 'Aqtau' issue:
    my($deltaz) = $zoff - $kept_zoff;
  if( &CloseEnough($tran + $deltaz, $kept_tran) ) # [smarter than == ]
    { # Once adjusted for zone offset change, we have a collision:
      $tran = $kept_tran;
    };
    # end 'Aqtau'.

    my($isdup) = ( ($kept_tran == $tran)
               &&($kept_dst == $dst)
               &&($kept_zoff == $zoff)
               );
  if($isdup)
    { Timely::XPrint( 0, "\n  Redundant tz entry $year: (" . &JulianDay($tran)
                . ") (d=" . &Tim($dst) . ';z=' . &Tim($zoff) . ") zone=$zone, "
                . &JulianDay($YearBot) . " -- " . &JulianDay($YearTop)
                . ' dst max=' . &Tim(MAXDST) . ' z max=' . &Tim(MAXZOFF) );
    }
```

```
 elsif( ($tran < $YearBot)
       ||($tran > $YearTop)
##     ||($dst < 0)  ## NO! $dst can be < 0 [explore in detail 4/12/2020]
       ||( abs($zoff) > MAXZOFF) # [ideally need test rtn]
       ||( $dst > MAXDST)        # [likewise]
      )
   { Timely::XPrint(0, "\n       *DEFECTIVE* timely entry suppressed: "
       . ' d=' . &Tim($dst) . ' max=' . &Tim(MAXDST)
       . ' z=' . &Tim($zoff) . ' max=' . &Tim(MAXZOFF)
       . " $mod/$type> ");    # [consider forcing an error]
   } else # not defective:
```

Not defective:

```
   { ($kept_tran, $kept_dst, $kept_zoff) = ($tran, $dst, $zoff);
     my($HTRAN)= &MegaDate($tran); # convert to big integer, from 'Julian day' value
     my($HDST) = &MegaDate($dst);
     my($HZOFF)= &MegaDate($zoff);

   if($SingleZone)
     { print " !" . &JulianDay($tran) . "(" . &Dat($tran) . ') d=' . &Tim($dst)
           . ' z=' . &Tim($zoff) . " $mod/$type" ;
     } else  # currently do NOT alter SQL if single zone:
     {
       my($qq) = "SELECT zone_offset, dst FROM timely "
               . "WHERE region=$region and transition=$HTRAN";
         ## Do *not* include year, allows duplicate transitions at year borders
         ## . "WHERE region=$region and year=$year and transition=$HTRAN";
       my($oz, $od) = Timely::GetSQL($handDB, $qq, 'ensure no duplicate');
       # NB. these values are microseconds.

       if(defined $oz)
         { my($HOZ) = Timely::HugeToJ($oz);
           my($HOD) = Timely::HugeToJ($od); # convert to Julian fragment
         if(  &CloseEnough($HOZ, $zoff)
           && &CloseEnough($HOD, $dst)
           ) ## if concordant
           { &TimeWarn('Duplicate timely entry (' . &Tim($zoff) . ', '
                     . &Tim($dst) . ") " ## . "as <$qq>"
             . &Tim( Timely::HugeToJ($oz) ) . ', ' . &Tim( Timely::HugeToJ($od) ),
                     55, $zone, $year);
           } else # discordant
           { Timely::XPrint(0, "\nIGNORED discord: Z=" . &Tim($zoff) . ' ' . &Tim($HOZ)
                     . '; DST=' . &Tim($dst) . ' ' . &Tim($HOD)
                     . "; zone=$REZONE{$region} ($region), transition=" . &Dat($tran)
                    );
            # These do not impact on existing transitions. Do NOT translate/amend!
           };
         } else # not duplicate
         { my($tkey) = Timely::FetchKey('timely', 'timely code');
```

```
          my($qi) = "INSERT INTO timely (timekey, region, year, transition, "
                   . "zone_offset, dst, ignored)"
           . "VALUES ($tkey, $region, $year, $HTRAN, $HZOFF, $HDST, $IGNORED)";
           # [? ver, chk]
        if( Timely::DoSQL($handDB, $qi, 'insert row') < 0)
          { &Aagh("failed to insert row ($qi)", __LINE__, 0);
            return;
          };
        }; # end else (not duplicate)
      };  # end else (not single zone)

    };  # end else (not defective)
  };  # end foreach
}
```

#### 5.8.2.2 Close enough

Ensure two numbers are within an 'epsilon' of one another. The submitted values should be Julian day numbers or fragments of a day.

```
sub CloseEnough #
{ my($J, $K) = @_;
  return( abs($J-$K) < EPSILONSECONDS );
}
```

#### 5.8.2.3 Big Julian number

Formerly converted Julian day number to microseconds; now simply multiply by 1M.

```
sub MegaDate #
{ my($d);
    ($d)=@_;
if(! defined $d)
  {  # [warn here?]
    return(0);
  };
if( length($d) < 2)   # '0' will match, as will ''
  { # [warn here?]
    return(0);
  };

if ($d !~ /^-?\d+\.?\d*$/ ) # at present don't allow leading/trailing space.
   { Timely::XPrint(0, "\nBad number for Julian day in seconds '$d'");  # [? &Aagh_()]
     &LogError("Bad number for Julian day '$d'"); # [Who will read the log?]
     return(0);
   };

  return($d*1000000); # KISS.
}
```

#### 5.8.2.4 Running zone

The first five arguments are all values in seconds; the last is Boolean.

If current cutoff Julian time (`$j`) is within this year (between `$lo` and `$hi`), return the zone offset supplied in the first parameter. Otherwise, the basic idea is that if it's earlier, return the current value (in `$ZO`); if it's later, then return the value that obtains after the end of the year, as this extends backwards in time to affect the current year (Zone rules work backwards). There is a catch, in that once we've encountered a rule subsequent to the current year, then we do NOT wish to overwrite this with any later rules, which can never apply. Hence the clumsy use of the Boolean value `$TOPPED`, which tops off the value.

In other words, the value returned is always the most recent zone offset that applies before or to the current year.

```
sub TryZone #
{ my($GMTOFF, $j, $lo, $hi, $ZO, $TOPPED) = @_;

if($j >= $lo && $j < $hi)
  { #&Debug(0, "\n==> ZO changed from " . &Tim($ZO) . " to " . &Tim($GMTOFF));
    return($GMTOFF, 0);
  };
if( ($j < $lo)
  || $TOPPED
  )
  { #&Debug(0, "\n==> ZO left at " . &Tim($ZO) );
    return($ZO, $TOPPED);
  };
  ## if $TOPPED NOT set AND $j >= $hi
    #&Debug(0, "\n==> ZO finalised from " . &Tim($ZO) . " to " . &Tim($GMTOFF) );
  return($GMTOFF, 1);
}
```

#### 5.8.2.5 Simple "between"

The lower limit is included, the upper, excluded. Alter zone transition if in the transition period, otherwise return the old value.

`$ZO` is used to track current zone offset, whatever.

```
sub Between #
{ my($zone_transition, $j, $lo, $hi) = @_;
if($j >= $lo && $j < $hi)
  {
    #&Debug(0, '<yes>');
    return($j);
  };
  #&Debug(0, '<no>');
  return($zone_transition);
}
```

## 5.9 Apply rule

Only invoked from within ZoneYear (5.8). Given the 'name' of a rule in `$rnam` (which may simply be an offset — this we ignore), determine whether the rule applies to the `$zone` for this year (`$yr`). Invokes Store () as appropriate.

We also supply the current zone offset, and the carried over DST from the preceding year (or zero if there was no carry over). A clumsy revision of my clunky original. A good source of rule specifications is the file *zic.8.txt* located within the *tzcode20???.gz* zipped file obtained from IANA. For a fair overview, see http://www.cstdbill.com/tzdb/tz-how-to.html.

1. Identify all rules that apply to this year (`$yr`) from the rule set in `$rnam`. Remember that rules apply *prospectively*, unlike transitions.
2. Actually apply the rules to generate daylight saving timestamps: a start time, an end time, and a DST offset. This is made difficult because:
   (a) Starting and ending rules may have different years;
   (b) The specified day for this year can be something like "The first Sunday *after* the 7th of August", or even *before* a date.
      i. The "first" can wrap around to a succeeding or even back to a previous month.
   (c) Reference times may be GMT, local time (excluding currently applicable DST), or "wall time" (which includes current DST).
   (d) DST can be carried from one year to the next.
   (e) There may potentially be multiple episodes of "turning on" and "turning off" of DST within a given year.

The `$zone_transition` variable is interesting as you'd think that any rule can only logically apply between the preceding transition and the current transition in this variable. For example, if a zone line specifies C-Eur applies up till 1945, and the preceding transition is 18 April 1941 at 23:00 (wall time), then when we apply a rule to 1941, we'd expect the C-Eur rules for 1941 only to apply after 18 April. There is however a catch. DST in this instance was established on April 1 1940 (at 2:00s) and not repealed until November 2 1942 at 2:00s.

### 5.9.1 A further problem

There's yet another issue. When we apply a DST-changing rule, the wall time will be affected. It is common to determine transition times based on wall time, so after we've changed DST, the relationship between any "UTC" time and the specified wall time will change. Now if you go back to ZoneYear (5.8) and examine it carefully, it extracts one or more zone rules and applies them. Each such rule will contain a transition time—and this was determined using the "then" DST and zone offset values.

If these change, we need to amend the subsequent transition times! The invoker can do this, simply knowing the old and new DST values. We thus return the value `$RUNNING_DST`, supplied as `$DSTCARRY`.

### 5.9.2 Yet more

There's another issue, exemplified by Asia/Gaza around 27 August–1 September. A simple zone rule applies until 1 September 00:00 wall time, but an incautious subsequent invocation of Palestine creates a spurious short-lived, apparent rule just before the termination of the zone rule! This can however be trapped by Store (), which will only store a rule if the timestamp provided is after `$LASTTRAN`.

### 5.9.3 Return values

Two values are returned:

**$STOREDOK** The result of the most recent Store () invocation. If none occurred, then zero. Note that Store () can return a value of zero, if nothing was stored.

**$RUNNING_DST** As above, the current DST that prevails on leaving. This starts off as `$DSTCARRY`, but can take on a new value including zero.

### 5.9.4 Code

Submitted parameters:

**zone** name of zone

**yr** year eg. 2001

**rnam** the rule name

**DSTCARRY** DST carried in to rule, in seconds

**zone_transition** set to a transition value if a transition is present, in seconds

**LASTTRAN** prior transition [explore, notably the value in `$DSTCARRY`], in seconds

**CURRENT_ZO** the zone offset that currently pertains, in seconds

**LASTZO** Most recent zone offset, in seconds

**SingleZone** Are we simply *debugging* a single zone? (1; otherwise 0).

We continually keep track of the current DST in `$RUNNING_DST`.

```
sub ApplyRule #
    # [the utility of the $LASTZO parameter seems doubtful: check me!]
{ my($zone, $yr, $rnam, $DSTCARRY, $zone_transition, $LASTTRAN, $CURRENT_ZO,
     $LASTZO, $SingleZone) = @_;

if($SingleZone)
  { print "[$rnam] ";
  };
  my($zoneinfo) = "$zone|$yr|$rnam";

  my($RUNNING_DST) = $DSTCARRY;
```

```
  my($STOREDOK) = 0;
  my($DELTA_Z) = $CURRENT_ZO - $LASTZO;  # find any change in zone offset
      #  from when original timestamp was determined. Must subtract from original
    my($dst_defer_start, $dst_defer_offset, $dst_defer_T) = (0,0,0); # see below
```

### 5.9.5 A rule

Look up the rule by name:

```
  my($rv) = $RULES{$rnam};  # assume exists
if( ref($rv) ne "ARRAY" )
    { Timely::XPrint(0,  "\n ERROR ***Bad definition*** for rule $rnam: ($yr) "
            . ref($rv) ) ;
      return(0,$RUNNING_DST);
    };
  my(@R) = @$rv;    # dereference
  my($r);

  my(%OFFSET) = (); # will associate timestamp with offset!
  my(%SUFFIX) = (); # used for TZ suffixes associated with a timestamp
  my(%CURRENT)= (); # only true if $yr WITHIN RANGE of specified rule, default false
  my(%IDS) = (); # for rule IDs
  # [explore zone time change/DST change alters timestamp order once suffix applied!?]
  # [this is theoretical rather than a concern]

  # an optimisation [see usage]:
  my($CARRYOVER) = 0;     # value is Julian seconds.
  my($CARRYsuffix) = '';
  my($CARRYsave) = '';
  my($CARRYid) = '';
```

### 5.9.6 Does the rule apply?

The main “for” loop that identifies all rules of interest.

```
for $r (@R)
  { my($FROM, $TO, $IN, $ON, $AT, $SAVE, $tz_rule) = @$r;
    my($RULETEXT) = "\n  Testing rule[$tz_rule]: "
                  . "$FROM, $TO, $IN, $ON, $AT, $SAVE-->"; # debug only
    my($VALID) = 1; # default to 'ok' for each rule.

  if($SingleZone)
    { # Timely::XPrint( 0, "($RULETEXT" );
      Timely::XPrint(0, '(' );
    };

  if($yr < $FROM)  # if year of interest is BEFORE the FROM of the rule
    { #&Debug(0, "\n  --skip $rnam ($FROM)" );   # cannot apply here.
    } else
    {
    if($SingleZone)
```

```
    { # print ",";  # [hmm, needs refining]
    };
  if( ( ($TO eq 'only')  # if precisely the year of interest
      &&($yr == $FROM)
      )
    ||($TO eq 'max')     # OR all years
    ||( ($TO ne 'only')  # OR year of interest is in the range
      &&($yr <= $TO)
      )
    )
    { my($ts, $sfx) = &FixRuleDate($yr, $IN, $ON, $AT, $zoneinfo); # NO GPS-adjust
    if($ts == 0) # often just a date excursion:
      { Timely::XPrint(0, " Skipped rule[a]: $FROM, $TO, $IN, $ON, $AT, $SAVE "
                 . "in zone $zone : $yr, $rnam");
        # return(0,0);
        $VALID = 0; # prevent creation of rule..
      };
    if($VALID)
      { $OFFSET{$ts} = $SAVE;  # timestamp as key! $ts *seconds*(Julian) like $SAVE
        $SUFFIX{$ts} = $sfx;   # see suffix use below...
        $CURRENT{$ts} = 1;     # FLAG rule as relevant to CURRENT year
        $IDS{$ts} = $tz_rule;  #
      if($SingleZone)
        { print "$TO";
        };
      };
      my($sav) = 'OFF     ';
    if($SAVE != 0)           # numeric value, 0 signals OFF
      { $sav = &Pad( '+' . &Tim($SAVE),8 );
      };
      Timely::Log("\n   * rule $sav" . &GDat($ts) . " $rnam($yr) : "
          . "from=$FROM to=$TO $IN $ON $AT ");
      # not yet GPS-adjusted
```

Now here's a peculiar anomaly. I call this the "year minus one" problem. A rule may apply in the current year, but the logic may depend on its application in the preceding year as well!

```
    if($FROM < $yr)  # must apply to the preceding year
      { my($prior) = $yr-1;
        my($ts, $sfx) = &FixRuleDate($prior, $IN, $ON, $AT, $zoneinfo);  # "year-1"
      if($ts == 0) # error
        { Timely::XPrint(0, " Skipped rule[b]: $FROM, $TO, $IN, $ON, $AT, $SAVE "
                   . "in zone $zone : $yr, $rnam");
          $VALID = 0;
        };
      if($VALID)
        { $OFFSET{$ts} = $SAVE;     # save DST, seconds
          $SUFFIX{$ts} = $sfx;      # suffix
          $CURRENT{$ts} = 0;        # SIGNAL only for use in DST calculations
```

```
          $IDS{$ts} = $tz_rule;  #
        if($SingleZone)
          { # print "<";
          };
        };
        my($gr) = Timely::Greg($ts, 0, 0, 0); # is NOT GPS-adjusted
        Timely::Log( "\n _  PRIOR $rnam($prior) : $gr" ); #
      };
    } else  # NOT active but potentially relevant in terms of DST carry-over:
```

Things are a bit more tricky if the year is not current, but there is the possibility that daylight saving may need to be carried over from a previous year. Each such rule starts and ends before the current year (`$yr`) — we are interested in the last year in which the rule applied.

The use of `$CARRYOVER` allows us to limit this exception to the most recent such rule:

```
    { my($top) = $TO;  # CANNOT be 'max'
    if($top eq 'only')
      { $top = $FROM;
      };
      my($ts, $sfx) = &FixRuleDate($top, $IN, $ON, $AT, $zoneinfo);
    if($ts == 0) # error
      { Timely::XPrint(0, " Skipped rule[c]: $FROM, $TO, $IN, $ON, $AT, $SAVE "
                  . "in zone $zone : $yr, $rnam");
        $VALID = 0;
      };
    if($VALID)
      {
      if($ts > $CARRYOVER) # update to most recent timestamp: [explore]
        { $CARRYOVER = $ts;
          $CARRYsuffix = $sfx;
          $CARRYsave = $SAVE;
          $CARRYid = $tz_rule;  #
        if($SingleZone)
          { print "<-";
          };
        }
      elsif($SingleZone)
        { print "-";  # nice try, but..
        };
      };
    };
  };

if($SingleZone)
  { print ')';
  };
};
```

```
  # [in following ? ensure $OFFSET{$CARRYOVER} doesn't already exist? [explore]]
if($CARRYOVER > 0)
  { $OFFSET{$CARRYOVER} = $CARRYsave;  # use timestamp as key!
    $SUFFIX{$CARRYOVER} = $CARRYsuffix;   # see suffix use below...
    $CURRENT{$CARRYOVER} = 0;     # only interested in DST
    $IDS{$CARRYOVER} = $CARRYid;  #
  };
```

Exit if no match, otherwise initialise variables for the next section:

```
my($numhash) = scalar(keys %OFFSET);
if($numhash < 1)
  { #&Debug(0, '<nil>');
  if( $zone_transition && $RUNNING_DST ) # if transition, carry moved in, END OFF!
    { #&Debug(0, "\n  apply both OFF: t was " . &Dat($zone_transition)
      #         . ', d was ' . &Tim($RUNNING_DST) );
      return(0, 0);   # do NOT return $RUNNING_DST. [might check this]
    };
    #&Debug(0, "\n  apply t OFF: t was" . &Dat($zone_transition)
    #         . ', d=' . &Tim($RUNNING_DST) );
    return(0,0); # [odd]
  };

  print '.'; # eye candy says "one or more rules are being processed..."
  # process timestamps in ascending order, to obtain DST start, end.
  # This is important as wall clock time depends on preceding DST :-(
```

### 5.9.7 Process rules in order

Process timestamps in ascending order. We determine `$Tm`, a GPS timestamp that describes the time contained in each rule.

The DST offset for each timestamp (seconds) is stored in `%OFFSET`; the suffix that determines modification of the timestamp (wall time, local time or GMT) is stored in `%SUFFIX`, and whether the line is "historical" (only used to determined carried-over DST from past years) or current (applies to current year) is contained in `%CURRENT`.

```
  my($ts);
  my($rulecount) = scalar %OFFSET;
  Timely::Log("\n\n  Applying $rulecount rules ($SingleZone):");

SORTED: foreach $ts (sort keys %OFFSET)  # sort timestamps ASC (various formats! )
  { my($De) = $OFFSET{$ts};
    my($suff) = $SUFFIX{$ts};
    my($Tm) = &FixBySuffix($suff, $ts, $CURRENT_ZO, $RUNNING_DST);
              # adjust TO GMT, seconds ($ts is in seconds too)

    my($tz_rule) = $IDS{$ts};

  if($SingleZone)
    { print "[$suff," . &Dat($Tm);   # [hmm]
```

```
    };

    # THERE'S A POTENTIAL PROBLEM HERE. For example, $zone_transition was calculated
    # previously and may have had a different DST applied, but later we depend on
    # a comparison between $Tm and this!
    my($DELTA_DST) = $RUNNING_DST - $DSTCARRY; # ? DST changed since zone_transition ..
    $Tm = Timely::ApplyGps($Tm);   # convert to GPS time
              # relevant, as $suff may implicate current wall clock time!
```

In the above, the calculation of `$Tm` the Julian day of the adjusted timestamp is pivotal. Next, we consider the rules.

Debugging:

```
    my($fancy) = '_';   # only relevant to carried over DST
  if($CURRENT{$ts})
    { $fancy = '*';     # rule is applicable to this year, in all its splendour
    };
```

#### 5.9.7.1 An anomaly

There is a problem typified by the rules for Europe/Belgrade in 1941. A new rule applies only after a transition, but it's possible that parts of the new rule extend backward in time, with daylight saving having been carried over for several years. In the example, up to the transition point on Friday 18 April at 11pm, there was no DST, but we then apply the C-Eur rule. In this rule, looking retrospectively, DST was turned on in 1940 but only removed in 1942, so DST must be active at the transition point.

I thus record rules that applied prior to the current year. This is tricky. Consider for example Zone=America/St_Johns. The daylight saving rules (and there are many) start and end at different times, overlapping in ways that mean you can't simply work forward from the first to the last start date.

Even this is however not sufficient, for the following reason, exemplified by St John's. An "on" rule may terminate in the year prior to the year of interest, but the "off" rule may persist to the year of interest. Because we don't take this "off" rule into account in our determination of DST carrying, we falsely assume that DST is still active, whereas it was turned off. This is the rationale for the "year minus one" code in Section 5.9.6.

All values stored are Julian but *not* yet adjusted for GPS, DST or zone offset. The code is very finicky.

### 5.9.8 A prior rule

If the rule we're looking at doesn't apply to the current year, but is merely an exercise in looking back to see whether DST is turned on, then turn on or off, storing the "legacy values":

```
  if(! $CURRENT{$ts} )  # if only looking at carry over of DST...
    { # do nothing
    if($SingleZone)
      { print ':'; # [use a better indicator?]
      };
    }
```

### 5.9.9 Too early

A previous transition point has already been passed:

```
    elsif
     ( ($Tm + $DELTA_DST - $DELTA_Z < $LASTTRAN ) # applies before _prior_ transition
     )
     { Timely::Log('< *skip, before prior transition ' &Tim($DELTA_DST)
       . '>' );   # [?DELTA_Z]
     if($SingleZone)
       { print "<";
       };
     }
```

### 5.9.10 A current rule

Otherwise, give it the works — the rule is “active”.

```
    else  # is CURRENT rule:
    {
```

There’s still a residual problem, exemplified by America/Montevideo 1942. We end off the “old dst carry over check” above with the legacy DST on at the time of entering the target year. But we need to turn it off using a rule that applies in the current year, if before the transition time. This is done below (“–DST disabled”), simply setting `$RUNNING_DST` = 0 if `$De` is 0 before the transition and `$tran` is set.

### 5.9.11 Too late

Now for our options. The first is:

- we’re in a transition year (`$zone_transition` is nonzero);
- but `$LASTTRAN` is zero — we’re not yet examining a rule that applies *after* the transition; and
- the `$Tm` value is after the transition point, so we ignore this rule, as it doesn’t yet apply.

Let’s explore this:

```
#   &Debug(0, "\n      TESTING T=" . &Dat($Tm) . ', tr=' . &Dat($zone_transition)
#           . ', delta=' . &Tim($DELTA_DST) );
    if( $zone_transition
      && ($Tm + $DELTA_DST - $DELTA_Z > $zone_transition)
      )     # rule can only apply < tr
      { Timely::Log(" *SKIP to END($rnam : $yr " . &Tim($De) . ") as "
           . &Dat( $Tm + $DELTA_DST )  . " adjusted by " . &Tim($DELTA_DST)
           . ' and ' . &Tim($DELTA_Z)
           . " is above transition ". &Dat($zone_transition) );
      if($SingleZone)
        { print ">";
```

```
        };
        $rulecount --;
        last SORTED;  # and finished as all others will be above too! [EXPLORE]
      }
```

#### 5.9.12 Just right

Otherwise, the rule applies. We may be turning DST on or off.

##### 5.9.12.1 The end of DST

Final else within main loop:

```
    else
      { #&Debug(0, "\n      Apply: ");
      if($De == 0) # if end of daylight saving  [END OF DAYLIGHT SAVING]
        { Timely::Log('End DST.');
      if($SingleZone)
        { print "\n  (-)"; # [ugh]
        };
```

The end of DST is problematic for *established* timestamps based on wall time—as they will need to have the daylight saving surgically excised :)

##### 5.9.12.2 Start of DST

Alternatively, `$De` is nonzero, we're starting DST.

```
        } else   # [start of DST]
        { Timely::Log('Start DST');
          my($plus) = '+';
        if($RUNNING_DST)
          { Timely::Log(" **NOTE** super DST: " . &Tim($RUNNING_DST)
                 . ' -> ' . &Tim($De) ) ;
            $plus = '++';
          };
        if($SingleZone)
          { print "\n  ($plus)";
          };
```

Else start DST:

```
        };  # [end "start of DST" section!]
          $STOREDOK = &Store($LASTTRAN, $Tm, $RUNNING_DST, $CURRENT_ZO, $suff,
                             DSTTRAN, 0, $SingleZone); # seconds
      };
    };   # end of final HUGE else within SORTED loop

    $RUNNING_DST = $De;
    #&Debug(0, "  runDST7=" . &Tim($RUNNING_DST) );

  if($SingleZone)
```

```
    { print "]";
    };
    $rulecount --;
  };   # END of SORTED loop.
```

### 5.9.13 Done

Exit:

```
if($STOREDOK > 0)
  { $STOREDOK += 7200;  # add 2h, just in case [hack]
  };
  #&Debug(0, "\n  apply returns: $STOREDOK, t=" . &Dat($STOREDOK)
  #        . '; d=' . &Tim($RUNNING_DST) );
  return($STOREDOK, $RUNNING_DST);
}
```

### 5.9.14 Subsidiary DST routines

#### 5.9.14.1 Pad

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

#### 5.9.14.2 "Dat" &c

This is a simpler version of Greg (). We assume the supplied value is GPS adjusted, but do *not* perform daylight saving or zone adjustments!

```
sub Dat # Submitted Julian value is in seconds not days
{ my($J) = @_;
  return( Timely::Greg($J,0,0,1) );
}
```

In the following variant, also in seconds, we *also do not* assume GPS adjustment:

```
sub GDat #
{ my($J) = @_;
  return( Timely::Greg($J,0,0,0) );
}
```

#### 5.9.14.3 Fix by suffix

Given a 'raw' timestamp, depending on the character value in $suff, perform one of the following:

**w** Adjust from local wall clock time to GMT;

**s** Adjust from local standard time (excludes any DST which might be present) to GMT;

<**nothing**> Assume wall clock time, and adjust accordingly as for 'w'

<**anything else**> Assume GMT, simply return the time value. This corresponds to **g**, **u** and **z**.

Things are made more complex by the possibility that the GMT offset of a local timestamp and indeed the daylight saving time adjustment might depend on whether it's above or below a transition point (`$zone_transition`). The interesting thing is that the comparison depends on adjustment of the timestamp for existing DST and GMT offset!

Logically, if the resultant timestamp is above a transition, then it must be adjusted by:

- The new DST that applies after the transition;
- The zone offset that applies after the transition.

This is tricky, as we'll sometimes be called on to adjudicate a timestamp that is close to the transition, without knowledge of the new offset

All of `$T`, `$GMTOFF` and `$newdst` are in seconds, not Julian days or parts thereof.

```
sub FixBySuffix #
{ my($suff, $T, $GMTOFF, $newdst) = @_;

  my($naah) = 0;
#  &Debug($naah, "  `$suff:" . &Dat($T)
#        . ', z=' . &Tim($GMTOFF) . ', new DST=' . &Tim($newdst)
#        . '--->' );
if($suff =~ /[guz]/)
  { #&Debug($naah, "gmt`" );
    return($T);
  };

if($suff eq 's') # standard local time (excluding daylight saving)
  { $T -= $GMTOFF;
    #&Debug($naah, "local:" . &Dat($T) . "`" );
    return($T);        # time has GMT offset added in, so subtract it
  };

if( (length $suff == 0)
  ||($suff eq 'w') # wall clock time
  )
  { $T -= $GMTOFF;
    $T -= $newdst;
    #&Debug($naah, "wall:" . &Dat($T) . "`" );
    return($T); # both DST and GMT have been ADDED
  };
```

```
  die "...\n*ERROR* Bad suffix ($suff) for time" . &JulianDay($T);
  return(0);
}
```

We have a substantial problem in the above — different GMT offsets may well apply before and after the timestamp specified in `$zone_transition`. Now the latter value is specified as a proleptic GPS time, but the value in $T may be (and usually is) a wall clock time. Even if it's a "standard local time" the GPS adjustment will differ depending on whether the value is after or before the transition.

To make the comparison, we thus need to make the same adjustments we did to `$zone_transition`. This involves:

1. subtracting `$BIAS`;
2. subtracting `$DSTCARRY`;
3. GPS adjustment by a few seconds.

#### 5.9.14.4 Fix time

Given a time as hh:mm:ss, hh:mm or hh, convert to seconds.

```
sub FixTime #
{ my($t) = @_;
  my($sign) = 1;
  ## my($naah) = 1;
  ## &Debug($naah, "<<$t ~ " );

if($t =~ /^-/ )
  { $sign = -1;
  };               # [this whole rtn is clumsy]

  my($r);
if($t =~ /(\d+):(\d+):(\d+)$/ )   #  [hours, minutes, seconds]
  { return( $sign * ($1*3600 + $2*60 + $3) ); # NB. sign distributes over the lot
    ## $r = $sign . ($1/24 + $2/(24*60) + $3/(24*3600));
    ## &Debug($naah, "$r>> " );
    ##return (  $r  );
  };
if($t =~ /(\d+):(\d+)$/ )  # [hours, minutes]
  { return( $sign * ($1*3600 + $2*60) );
    # $r = $sign . ($1/24 + $2/(24*60));
    # &Debug($naah, "$r>> " );
    # return (  $r  );
  };
if($t =~ /(\d+)$/ )  # [hours]
  { return( $sign * $1*3600 );
    # $r = $sign . ($1/24);
    # &Debug($naah, "$r>> " );
    # return (  $r   );
  };
```

```
  die "\n***ERROR bad time: $t";
}
```

#### 5.9.14.5 Fix rule date

This routine deals with anomalies like “lastSun” and “Sun>=1”. The rule …

```
    Rule Zion 2005 2012 - Apr Fri<=1 2:00 1:00 D
```

… is a problem! This is the only usage in tz of “<=”; the rubric in the file *asia* explains that:

> The proposed law agreed upon by the Knesset Interior Committee on 2005-02-14 is that, for 2005 and beyond, DST starts at 02:00 the last Friday before April 2nd (i.e. the last Friday in March or April 1st itself if it falls on a Friday) and ends at 02:00 on the Saturday night _before_ the fast of Yom Kippur.

The values returned are a Julian value in seconds, and a text suffix.

```
sub FixRuleDate #
{ my($yr, $MMM, $dd, $hhmm, $zoneinfo) = @_;
  my($mo) = &FetchMonth($MMM);
if($mo < 1)
  { Timely::Warn(3, "BAD RULE month: $yr m=$MMM($mo) $dd $hhmm");
    return(0,0);
  };

  my($aday) = $dd;

  # last of month eg 'lastSun'
if( $dd =~ /^last(\w+)$/ )
  { my($wday) = $1;
    $aday = &FetchLast($yr, $mo, $wday);
  }
  # MMM>=n e.g. 'Sun>=1'
elsif( $dd =~ /^(\w{3})>=(\d+)$/ )
  { ($aday, $mo) = &FetchFirst($yr, $mo, $1, $2);
  }
elsif( $dd =~ /^(\w{3})<=(\d+)$/ ) # only instance is Zion !
  { # Timely::Warn(0, "Testing <= [Zion]");
    ($aday, $mo) = &FetchBefore($yr, $mo, $1, $2);
  };
  # $aday now should be numeric:
if( $aday !~ /^\d+$/ )
  { Timely::Warn(3, "Failed date criterion($yr, $MMM .. $hhmm): '$dd'");
    return(0, 0);
  };

  my($hh);
  my($mm);
```

```
  my($ss) = 0;
  my($SUFFIX) = '';
if($hhmm =~ /^(\d+):(\d+):(\d+)([guzsw]*)$/ ) # clumsy
  { $hh = $1;
    $mm = $2;
    $ss = $3;
    $SUFFIX = $4;
  }
elsif($hhmm !~ /^(\d+):(\d+)([guzsw]*)$/ )   # technically ([guzsw]?) is better
  {
  if($hhmm =~ /^0$/ )
    { $hh = 0;
      $mm = 0;
      Timely::Log(" --assumed time '0:00' $yr $mo $aday <$hhmm>", 10);
    } else
    { Timely::Warn(3, "BAD hours/min ($zoneinfo) $yr $mo $aday $hhmm");
      return(0,0);
    };
  } else
  { $hh = $1;
    $mm = $2;
    $SUFFIX = $3;
  };
  my($J) = &ZoneJulian($yr, $mo, $aday, $hh, $mm, $ss); # seconds
  return($J, $SUFFIX);  # NB. $J is *not* GPS-adjusted!
}
```

#### 5.9.14.6 Fetch last

Given the name of a day, fetch that last day (e.g. Sunday) for the given year and month, as a Julian value.

```
sub FetchLast #
{ my($yy, $mm, $day) = @_;
  my(@MONS) = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
    # what about leap years? [???????????????? fix me]
  my(%WD) = ('Sun', 0, 'Mon', 1, 'Tue', 2, 'Wed', 3, 'Thu', 4, 'Fri', 5, 'Sat', 6);
if(! defined $WD{$day})
  { die "\n***ERROR*** Bad weekday: $yy $mm $day";
    return(0);
  };
  my($daycode) = $WD{$day};
if(! defined $MONS[$mm-1])
  { die "\n***ERROR***Bad month: $yy $mm $day";
    return(0);
  };
  my($topday) = $MONS[$mm-1];
  my($Jtest) = Timely::GpsJulian($yy, $mm, $topday, 0, 0, 0, 0, 0, 0)/86400;
  my($mod) = int($Jtest+2) % 7;   # modulo seven. +2 TO MAKE Sun=0
while($mod != $daycode)
  { $topday --;
```

```
    $mod --;
   if($mod < 0)
     { $mod = 6; # wrap [check this always exits, hmm]
     };
  };
  return($topday);
}
```

#### 5.9.14.7 Fetch first

Similar to fetch last, but identifies the *first* instance of a particular day (e.g. Sunday) *after or on* a given day of the month (e.g. the 15th). If the day is after the end of the month, segue into the following month!

This necessitates returning two values—both month and day.

```
sub FetchFirst #
{ my($yy, $mo, $day, $cut) = @_;
  my(@MONS) = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
  if( &IsLeapYear($yy) )
    { $MONS[1] = 29;
    };
  my(%WD) = ('Sun', 0, 'Mon', 1, 'Tue', 2, 'Wed', 3, 'Thu', 4, 'Fri', 5, 'Sat', 6);

if(! defined $WD{$day})
  { Timely::Warn(3, "BAD weekday: $yy $mo $day");
    return('', $mo);
  };

  my($wantday) = $WD{$day};
if(! defined $MONS[$mo-1])
  { Timely::Warn(3, "BAD month: $yy $mo $day");
    return('', $mo);
  };

  my($topday) = $MONS[$mo-1];
  my($Jtest) = Timely::GpsJulian($yy, $mo, $cut, 0, 0, 0, 0, 0, 0)/86400;  # first day
  my($dayofweek) = int($Jtest+2) % 7;   # modulo seven. +2 TO MAKE Sun=0 : CURRENT DAY
  #&Debug(0, "$Jtest d=$cut dow=$dayofweek ");

  # We now need to adjust $cut upwards, until it meets the right day.
  # corresponds to incr $dayofweek commensurately until matches $wantday(0=sun).
  # (The catch is that we may need to wrap $dayofweek around);
  # We can also wrap into the next month
  my($delta) = $wantday - $dayofweek; # desired day - current e.g. Wed-Sun = 3-0 = 3.

if($delta < 0)  # will have to wrap around e.g. Sun-Wed = 0-3 = -3, i.e. add 7.
  { $cut += (7+$delta);
  }
elsif($delta > 0) #
  { $cut += $delta;
  }; # else $cut is correct : the same day!
```

```
  # check that cut isn't ridiculous:
if($cut > $topday)
  { ## Timely::Warn(0, "FetchFirst($yy-$mo-$day) wrapped $cut after month ($topday)!");
    $cut -= $topday;
    $mo ++;
  if($mo > 12)
    { Timely::Warn(3, "FetchFirst($yy-$mo-$day) can't advance date past year end($topda
      return('', $mo);
    };
  };

  return($cut, $mo);
}
```

#### 5.9.14.8 Leap year?

Crude: A leap year if divisible by 4 unless divisible by 100 unless divisible by 400.

```
sub IsLeapYear
{ my($yy);    # clumsy, use @
    ($yy)=@_;
if(($yy % 4) != 0)
  { return(0); # not a leap year, not divisible by 4.
  };
if(($yy % 100) != 0)
  { return(1); # is a leap year
  };
  return(($yy % 400) == 0); # is a leap year if divisible by 400.
}
```

### 5.9.15 Fetch before

Implementing `$day <= $cut` : this can (and does) wrap back to the preceding month. Note the potential for sharing code (arrays) etc. with FetchFirst ().

NB. At present we do *not* wrap around back to the preceding year, we don't allow e.g. <1 January.

```
sub FetchBefore #
{ my($yy, $mo, $day, $cut) = @_;
  my(@MONS) = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
  ##
  if( &IsLeapYear($yy) )
    { $MONS[1] = 29;
    };
  my(%WD) = ('Sun', 0, 'Mon', 1, 'Tue', 2, 'Wed', 3, 'Thu', 4, 'Fri', 5, 'Sat', 6);

if(! defined $WD{$day})
  { Timely::Warn(3, "BAD weekday: $yy $mo $day");
```

```
    return('', $mo);
  };

  my($wantday) = $WD{$day};
if(! defined $MONS[$mo-1])
  { Timely::Warn(3, "BAD month: $yy $mo $day");
    return('', $mo);
  };

  my($Jtest) = Timely::GpsJulian($yy, $mo, $cut, 0, 0, 0, 0, 0, 0)/86400;  # last day
  my($dayofweek) = int($Jtest+2) % 7;   # modulo seven. +2 TO MAKE Sun=0 : CURRENT DAY

  # move back
  my($delta) = $dayofweek - $wantday ; # eg. Tue wants Sunday = 2-0

if($delta < 0)  # will have to wrap back e.g. Sunday wants Tuesday = 0-2
  { $cut -= (7+$delta);
  }
elsif($delta > 0) #
  { $cut -= $delta;
  }; # else $cut is correct : the same day!
  # Timely::XPrint(0, " Debug: cut is $cut,");

  # check that cut isn't bad:
if($cut < 1)
  {
  if($mo < 2)
    { Timely::Warn(3, "FetchBefore($yy-$mo-$day) can't move date before yr start($cut)"
      return('', $mo);
    };
    # Timely::Warn(0, "FetchBefore($yy-$mo-$day) wrapped $cut back!");
    my($lasttopday) = $MONS[$mo-2];
    $cut += $lasttopday;  # add negative value or zero (0=last day of previous month)
    $mo --;
  };

  return($cut, $mo);
}
```

### 5.9.16 Parse a TZ rule

This uses the global %RULES. Here the daylight saving time is converted to a numeric value using FixTime (5.9.14.4).

```
sub ParseRule #
{ my($ln) = @_;

  my($r, $NAME, $FROM, $TO, $TYPE, $IN, $ON, $AT, $SAVE, $LETTERS) = split /\s+/, $ln;

if(! defined $LETTERS)
  { Timely::Log( "\n?R=<$ln>\n" ) ;
```

```
    return(0);
  };
  my($tz_rule) = &tz_rule_save($NAME, $FROM, $TO, $IN, $ON, $AT, $SAVE);

  # here must check format of rule components, might restructure...

  my($saving) = &FixTime($SAVE); # seconds

  # end check.


  my (@V) = ($FROM, $TO, $IN, $ON, $AT, $saving, $tz_rule); # the key variables
    # note that $saving at index 5 is in seconds.

   # format is Rule NAME etc.
if(defined $RULES{$NAME} )
  { my ($R) = $RULES{$NAME};
    my (@A) = @$R;
    ## Timely::Log( " --existing rule,size=" . scalar @A );

# if we're in the N hemisphere, it's possible to have
#   end rules appear before start rules, which means
#   we can't compensate for the DST adequately. Fix this.
#   (You also can't simply sort by time, as rules have different extent!)
#   There is however a residual problem with 'super' DST (eg UK, Germany 1940s)
#########################################################
  if( ! $saving ) # if an END of DST rule
    { push( @A, \@V);     # push a reference
    } else
    { unshift( @A, \@V);  # put STARTing rules at start of array!
    };
#########################################################

    Timely::XPrint(0, 'r');
    $RULES{$NAME} = \@A; # and store as an array reference!
  } else
  { # Timely::Log(" --new rule--");
    my(@A) = ( \@V );   # create new array with reference
    $RULES{$NAME} = \@A;
    Timely::XPrint(0, 'R');
  };

  return(1);
}
```

The value stored in the associative array `%RULES` is a reference to an array. The latter array contains multiple sub-arrays, also stored as references.

I keep the “ENDing” rules at the end of the array so that when we parse rules we don’t inconveniently encounter an “end of DST” rule (with a local

timestamp that includes DST, D'Oh) before the starting rule — the ordering in TZ makes no such accommodation.

### 5.9.17 Start a new zone

Accepts a line and zone description.

```
sub StartZone #
{ my($ln, $zonedesc) = @_;

  my(@ZDAT) = split /\s+/, $ln;
  my($ZL) = scalar @ZDAT;
if($ZL < 6)
  { #  e.g. CET, WET etc
    Timely::Log( " --short zone data length=$ZL: <$ln> -- " );
  if($ZL < 5) # can't fix
    { die "\n***ERROR*** zone data too short <$ln>"; # [explore use of Aagh_()]
      return(0);
    };
    # here might compensate for missing final (end time) [? explore]
  };
```

Some more checks:

```
  shift(@ZDAT);  # discard 'Zone'
  my($NAME)   = shift(@ZDAT);
  my($SingleZone) = (length $zonedesc > 0);

  # if zone description exists, check this single zone
if($SingleZone)
  {
  if($NAME ne $zonedesc)    # and if no match:
    { return('');  # signal nil.
    };
    print "\n[Starting zone $zonedesc] ";
  };

  my(@V) = &ZoneX($SingleZone, @ZDAT);   # this should set %ZONES entry..

if(! defined $ZONES{$NAME} )
  { Timely::XPrint(0,  "\n *NOTE* Unknown zone: '$NAME', forcing:" );
    # create it, otherwise WET, CET, MET, EET, EST, HST, MST, EST5EDT ... absent
    # Also see the Montreal-related hack in ReadZones_()
    my($ZID) = &FetchCountryCode('UT');  # Rather use Universal Time than null 'XX'
    &SaveZone($NAME, $ZID, 0, 0); # also fills in ZONES, ZONECODES
  };
  my($zn) = $ZONES{$NAME};
if( ref($zn) eq 'ARRAY' )
  { die "\n***ERROR*** Duplicate zone: $NAME";
    return(0);
  };
```

Store, get identity of this zone row:

```
  my($zoneid) = $ZONECODES{$NAME};
  my($tz_cutoff) = &tz_cutoff_save($ZDAT[0], $ZDAT[1], $zoneid, $ZDAT[3]);
if(! defined $tz_cutoff || length $tz_cutoff < 1)
  { die "Missing zone code for '$NAME'";
  };
  # [$jvs: fix me! must now include $tz_cutoff ID in \@V]
  push(@V, $tz_cutoff);

  my(@A) = ( \@V );   # create new array with reference
  $ZONES{$NAME} = \@A;
  Timely::Log( "\n(Z:$NAME" );
  return($NAME);
}
```

Note that there are some Zones that have no UNTIL entry (e.g. EST, MST, EST5EDT etc) and these will force an error in the above.

### 5.9.18 Add to a zone

Given a line of data and the name of a zone, split the line into components, parse the data, and push the parsed data to an array indexed to the name of the zone in `%ZONES`. The value of `$SingleZone` is 1 iff we're dealing with just one zone.

```
sub MoreZone #
{ my($ln, $NAME, $SingleZone) = @_;

if(length $NAME < 1)  # Don't process.
  { return(1);
  };

  my(@ZDAT) = split /\s+/, $ln;  # here, first datum should be null string

if($SingleZone)
  { print " zone: " . join('|', @ZDAT);
  } else
  { print 'z';
  };

  my($ZL) = scalar @ZDAT;
if($ZL < 4)
  { &TimeWarn( "short zone $ZL=<$ln>\n", 11, $NAME, 0 );
    return(0);
  };

  shift(@ZDAT);  # discard first (null) element
  my(@V) = &ZoneX($SingleZone, @ZDAT);

if(! defined $ZONES{$NAME} )           # [might move this up higher?]
```

```
  { &TimeWarn( " (severe) missing zone:$NAME\n", 12, $NAME, 0 );
    # WHAT ABOUT DISCARDED DATA ? [explore]
    return(0);
  };

  # store, get identity for this row:
  my($tz_cutoff) = &tz_cutoff_save($ZDAT[0], $ZDAT[1],
                                   $ZONECODES{$NAME}, $ZDAT[3]);
  push(@V, $tz_cutoff);

  my ($R) = $ZONES{$NAME};
  my (@A) = @$R;
  push( @A, \@V);     # push a reference
  $ZONES{$NAME} = \@A; # and store as an array reference!
  # print 'z';
  return(1);
}
```

#### 5.9.18.1 Time-processing-specific warning

Increment warning count, and XPrint ():

```
sub TimeWarn #
{ my($msg, $id, $zone, $year) = @_;
  Timely::Warn(0, "-- [$id:$zone $year] $msg");
}
```

#### 5.9.18.2 ZoneX

ZoneX (), the zone extractor, is invoked by StartZone () and MoreZone (). As its name suggests, it extracts zone data from the supplied array `@ZDAT`. *Assumes* minimum length of `@ZDAT` of 4. $SingleZone is set to 1 if we're only interested in a single zone.

The "usual" format of `@ZDAT` is a GMT (Z) offset, followed by a rule (which may simply be '-'), then a formatting string like 'BMT' that is disregarded, followed by a terminal date that consists of at least a year, but also possibly a month (in a variety of formats), a day (optional) and a time. The time for the GMT offset and for the terminal date may be hh:mm or even hh:mm:ss. See for example Asia/Jakarta.

The return values:

**$GMTOFF** This is Z, the zone offset (in seconds)

**$RUL** The name of the rule for this zone row. The rule may be '-', signalling "No rule);

**$UNTIL** The year until which this applies, which may be absent, signifying forever (for now).

**$MMM,** $DD, $hh, $mm Month (0=absent) — used for debugging alone

**$FORMAT** — the "time format", usually empty, indicating wall time, but may be one of a variety of characters.

**$J** A Julian timestamp (end time) in seconds

**$suffix**

Unfortunately, there are many variants we have to deal with in the input data, for example we have 1995-Mar-lastSun in Asia/Aqtau!

The Julian value is the cutoff time, but it is not yet adjusted for region or DST, even if this is appropriate.

The invoker will inevitably store the returned values as an array in %ZONES, one of potentially many rows describing an individual zone. Later, ZoneYear () will extract these rows and use ApplyRule () to instantiate them as rows in the **timely** table.

```
sub ZoneX #
{ my($SingleZone, @ZDAT) = @_;
  my($lZ) = scalar @ZDAT;

  my($GMTOFF) = &FixTime($ZDAT[0]); #seconds
  # might here validate the zone time [explore]

  my($RUL)  = $ZDAT[1];
  my($FORMAT) = $ZDAT[2];   # will become last entry

  my($UNTIL)  = 9999;  # ??? [must accommodate missing value = ? "ad infinitum"]
if($lZ < 4)
  { Timely::Log("--short($lZ) zone data array: @ZDAT -- ");
  } else
  { $UNTIL = $ZDAT[3];
  };
  my($MMM)    = '0';
if($lZ > 4)
  { $MMM  = $ZDAT[4];
  };
  my($DD)     = '0';
if($lZ > 5)
  { $DD  = $ZDAT[5];
  };
  my($hhmm)   = '0:0';
if($lZ > 6)
  { $hhmm = $ZDAT[6];
  };
  my($hh)=$hhmm;
  my($mm)=0;
  my($ss)=0;
  my($suffix) = '';
if($hhmm =~ /(\d+):(\d+):(\d+)([guzsw]*)/ )
  { $hh = $1;
    $mm = $2;
    $ss = $3;
    $suffix = $4;  # clumsy
  }
```

```
elsif($hhmm =~ /(\d+):(\d+)([guzsw]*)/ )
  { $hh = $1;
    $mm = $2;
    $suffix = $3;  # clumsy
  };

  ######################################################
  my($mo) = &FetchMonth($MMM);
if($mo < 1)
  { die "\n***ERROR*** Bad rule date: $UNTIL $MMM $DD $hh $mm";
    return(0);
  };
```

A wrinkle:

```
  my($J);
    # here must accommodate e.g. lastSun
if( ($DD =~ /last/)
  ||($DD =~ /first/)
  ||($DD =~ /^\w{3}>=\d+/ )  # e.g. Sun>=
  )
  { ($J, $suffix) = &FixRuleDate($UNTIL, $mo, $DD, $hhmm, ''); # $J is in seconds.
                            # at present no zoneinfo here ^ [explore]
  if($J == 0) # error
    { die ("Bad rule in ZoneX: $RUL, $UNTIL, $mo, $DD, $hhmm, $FORMAT ");
           # not $FROM, $TO, $IN, $ON, $AT, $SAVE
      return(0,0);
    };
     #&Debug(0, "\n Fixing anomalous Zone Rule date: $UNTIL $MMM $DD ->" . &Dat($J));
  } else
  { $J = &ZoneJulian($UNTIL, $mo, $DD, $hh, $mm, $ss);  # seconds.
  };

if($SingleZone)
  { print "  : $GMTOFF, $RUL, $UNTIL, $MMM, $DD, $hh, $mm, $FORMAT, $J, $suffix\n";
  };

  # do NOT invoke Timely::ApplyGps_()
  return($GMTOFF, $RUL, $UNTIL, $MMM, $DD, $hh, $mm, $FORMAT, $J, $suffix);
}
```

#### 5.9.18.3 ZoneJulian

Convert zone timestamp data to a Julian value. If no year is provided, default to far in the future! The Julian value is in seconds, not days!

```
sub ZoneJulian #
{ my($UNTIL, $fm, $DD, $hh, $mm, $ss) = @_;

if(!defined $UNTIL)
  {$UNTIL = 9999;  # might also log warning [?]
```

```
  };
if(! defined $fm)
  { die "\n***ERROR*** Bad month for $UNTIL / $DD $hh $mm $ss";
    return(0);
  };
  #&Debug(0, "<zj $UNTIL $fm-$DD $hh:$mm:$ss>");

if($DD == 0) # artificial
  { $DD = 1;
  }
elsif($DD !~ /^\d+$/)
  { die "\n Bad Zone date: $UNTIL $fm [$DD] $hh $mm";
  };
  return( Timely::ToJulian($UNTIL, $fm, $DD, $hh, $mm, $ss, 0, 0, 0) );
}
```

In the above, we must *not yet* convert to GPS time, as there will be further adjustments before we do this!

#### 5.9.18.4 Fetch month

Next, Jan–Dec converted to 1–12; 0 to 1.

```
sub FetchMonth
{ my($mm) = @_;
  # permit numeric:
if($mm =~ /^\d+$/ )
  {
  if($mm == 0)  # turn 0 into 1
    { return(1);
    };
    return($mm);
  };  # simply return numeric.
my(%MTHS) = ('Jan', 1, 'Feb', 2, 'Mar', 3, 'Apr', 4, 'May', 5, 'Jun', 6,
    'Jul', 7, 'Aug', 8, 'Sep', 9, 'Oct', 10, 'Nov', 11, 'Dec', 12,
    0, 1, '0', 1); # 0-->1 !
if( ! defined $MTHS{$mm} )
  { return(0);  # fail
  };
  return( $MTHS{$mm} );
}
```

### 5.9.19 Terminate a zone

All this does is log the end of the zone by writing a right parenthesis. No checks are made of the ZONE, and it would theoretically be possible to add further zone data (although this would be unwise, and might even perhaps be checked for using a “closed zone” associative array [paranoid programmers only]).

```
sub EndZone #
{ my($ZN) = @_;
```

```
if(length $ZN < 1)
  { return(1);
  };

  Timely::XPrint(0,  ')' );

  return(1);
}
```

## 5.10 Testing zones

## 5.11 Zone test

In the menu invocation of ZoneTest (), the user has the option of selecting the default date by specifying "t" alone, or the month, day and time, which follow the "t". If a year is also specified, then this year and all years *after this year* are tested! The addition of `AND reason > -1` allows suppression of obsolete places.

Rather than wiring in some arbitrary value, an attractive tweak is to default to the current MM-DD hh:mm:ss displayed at the bottom of the menu made by SetMenu (), as this is visually consistent.

Testing always works *down* from the largest date in the **timely** table. The sequence of operations is:

1. Determine MM-DD hh:mm:ss. Options are:

   **t** alone defaults to the MM DD hh mm ss of the Julian value supplied in `$NUMBR`, taken as a GPS stamp, and converted to UTC Gregorian values.

   **t MM-DD hh:mm:ss** uses these supplied values

   **t YYYY-MM-DD hh:mm:ss** as above, use supplied values but only go down to year YYYY

   **t YYYY ZZZZ-MM-DD hh:mm:ss** between the years YYYY(lower) and ZZZZ, but use supplied values

   **t YYYY** uses 'defaults', but only go *down to* year YYYY (from topmost acceptable year)

   **t YYYY ZZZZ** defaults, but between the years YYYY and ZZZZ.

2. Check for silliness

3. Get a list of all active timezones from the **PLACES** table in ***f*ehr**:

4. For each year:

   (a) Convert Gregorian values to GPS Julian using Timely routine GpsJulian (-) on the numeric values for the full Gregorian date;

   (b) Convert Julian back to Gregorian using FullGregorian (-);

   (c) Check UTC value, using DateTime, establishing the `$dt1` object as a date parser;

(d) Then, *for each zone:*

i. Use J2G () to convert the *Julian* value to an appropriate Gregorian timestamp for this timezone.

ii. Do something similar in DateTime, using the `$dt1` object established above.

iii. Compare the two Gregorian values, and report any discrepancy.

It's important to understand the limitations of the above approach used to implement the **t** command, over and above the obvious issues that will arise if the version of DateTime doesn't match the version used in Timely. It's possible to create a wall time that is invalid, but that won't be picked up by the t screening mechanism, because every Julian value will have a UTC value and at least one corresponding Gregorian value, but the reverse doesn't necessarily hold. Contrast the use of **`t 10-05 00:00:00`** and **`w 10-05 00:00:00`**, for example. This motivates for the **w** command.

```
sub ZoneTest #
{ my($arg, $NUMBR, $myLowYear, $myTopYear);
    ($arg, $NUMBR, $myLowYear, $myTopYear)=@_;

  my($handDB) = $smalltime_h;

 my($stubyear,$MONTH,$DAY, $HOUR,$MINUTE,$SECOND)
               = Timely::FullGregorian($NUMBR, 0,        0,        1);
                            # no local ^, no DST ^, is GPS ^

if(length $arg < 1)
  { # do nothing, accept defaults..
    my($pd) = Timely::PrettyDate(0,$stubyear,$MONTH,$DAY, $HOUR,$MINUTE,$SECOND);
    print "\n Testing default: " . substr($pd, 4);
    ShowEscMessage();
  }
elsif($arg =~ /^\s*(\d{4})\s+(\d{4})-(\d+)-(\d+)[ T]+(\d+):(\d+):(\d+)\s*$/ )
  { $myLowYear = int($1) - 1;   # low and high years
    $myTopYear = int($2);
    $MONTH  = $3;
    $DAY    = $4;
    $HOUR   = $5;
    $MINUTE = $6;
    $SECOND = $7;
  }
elsif($arg =~ /\s*(\d{4})-(\d+)-(\d+)[ T]+(\d+):(\d+):(\d+)\s*$/ ) # + low yr
  { $myLowYear = int($1) - 1;
    $MONTH  = $2;
    $DAY    = $3;
    $HOUR   = $4;
    $MINUTE = $5;
    $SECOND = $6;
  }
```

```
elsif($arg =~ /^\s*(\d+)-(\d+)[ T]+(\d+):(\d+):(\d+)\s*$/ ) # just the month..
  { $MONTH  = $1;
    $DAY    = $2;
    $HOUR   = $3;
    $MINUTE = $4;
    $SECOND = $5;
  }
elsif($arg =~ /^\s*(\d{4})\s+(\d{4})\s*$/ ) # a range of years
  { $myLowYear = int($1) - 1;
    $myTopYear = int($2);
  }
elsif($arg =~ /^\s*(\d{4})\s*$/ ) # just the minimum year
  { $myLowYear = int($1) - 1;
  }
 else
  { print "\n Unknown format, expected MM-DD hh:mm:ss e.g. 12-31 20:30:00 ";
    return;
  };

if( ($myLowYear < $LOWYEAR)
  ||($myTopYear > $TOPYEAR)
  )
  { print "\n Bad year range $myLowYear .. $myTopYear ($LOWYEAR : $TOPYEAR-1)";
    return;
  };

  my($q) = "SELECT place, description FROM PLACES "
         . "WHERE p_amended between 2 and $TOPCOUNTRY AND reason > -1";
  #   ignore 1 = UTC stub.
  # place is internal code, description is zone code:
  my(@ZDAT) = Timely::SQLManySQL($handDB, $q, 'get zones');
```

For each year, test the date for every zone. I convert the specified date (for each year) first into a Julian day, and then back to a Gregorian one. The Gregorian date values are then used to construct a DateTime object, as DateTime uses the TZ database.

As the timestamps stored in our database are all based on GPS times, the value submitted to J2G () must be GPS-time based. When we convert this value back to Gregorian, we must thus also specify that the value is GPS-based.

```
  my($YEAR) = $myTopYear;
  my($ERR) = 0;
  my($ANOMALIES) = 0;

REDO: while($YEAR > $myLowYear)
  {
  if( CheckForEscape() )
    { $YEAR = $myLowYear-1;
      next REDO;             #
```

```
  };

  my($JD) = Timely::GpsJulian($YEAR,$MONTH,$DAY, $HOUR,$MINUTE,$SECOND, 0,0,0);
  my($xYYYY, $xMM, $xDD, $xh, $xm, $xs) = Timely::FullGregorian($JD, 0, 0, 1); #1:GPS

my($dt1) = eval { DateTime->new( year => $xYYYY, month => $xMM, day => $xDD,
     hour => $xh, minute => $xm, second => $xs,
     time_zone => 'UTC') };
if(! $dt1)
  { Timely::Warn(3, "INVALID time zone (DateTime usage) 'UTC'");
    last REDO;
  };
   Timely::XPrint(0,  "\n "
      . Timely::PrettyDate(0,$YEAR, $MONTH, $DAY, $HOUR, $MINUTE, $SECOND) . ' ' );
```

Now, for each zone, we compare the two dates — the Gregorian date obtained by applying J2G () to the Julian day, for that zone; and a cloned copy of the UTC date, adjusted for the same zone. We currently ignore "anomalous" dates, signalled by a return value of zero from J2G.

```
  my($ref);
foreach $ref(@ZDAT)
  {
    my($zid, $zone) = @$ref;
    my($YYYY,$MM,$DD,$h,$m,$s, undef,undef, $hog, undef, undef)
                # dummy values  ^    ^           ^      ^
           = Timely::J2G(0, $zid, $JD); # JD is Gps-based
  if($YYYY == 0)  # anomalous date
    { Timely::XPrint(0, "\n     'Anomaly' skipped: $zid/$zone $YEAR $JD");
      $ANOMALIES ++;
    } else
    { my($mine) = Timely::PrettyDate(1,$YYYY, $MM, $DD, $h, $m, $s);
      # similar to Timely::Greg_()

     # trap failure of DateTime [??]
     ## print "\ndebug .. '$zone'";

      my($dtest) = eval { $dt1->clone->set_time_zone( $zone ) };
    if(! $dtest)
      { Timely::Warn(3, "INVALID time zone (Perl DateTime test) $zone");
      } else
      { my($yours) = $dtest->datetime();  # format is identical
      if($mine ne $yours)
        { Timely::XPrint(0, "\n   Mismatch $zid/$zone $mine v $yours" . $TICK);
          $ERR++;
        } else
        { print '.';
          ## &Debug (0,"\n OK Zone $zone OK: $mine");
        };
      };
```

```
      };
    };
  $YEAR --;
  };
  Timely::XPrint( 0, "\n\n Errors=$ERR, $ANOMALIES anomalies for " .
          ($myLowYear+1) . '--'
          . Timely::PrettyDate(1,$TOPYEAR, $MONTH, $DAY, $HOUR, $MINUTE, $SECOND) );
}
```

It's important to validate a given timestamp across zones. Simplest is just:

1. Determine a list of all valid time zone names;
2. Establish a timestamp;
3. Determine YYYY MM DD hh:mm:ss for all zones using both our routines and DateTime.
4. Compare the two and log variances.

## 5.12 Wall test

This is similar to the above ZoneTest (5.10) but does the "opposite". Given a date and a region, in ***f*ehr** convert this to a UTC time; in the tz database, establish the date for that region and similarly convert to UTC; then compare the two UTC values! The addition of `AND reason > -1` allows suppression of obsolete places.

```
sub WallTest #
{ my($arg, $NUMBR, $myLowYear, $myTopYear) = @_;

  my($handDB) = $smalltime_h;

  # borrowed from ZoneTest_() :
  my($stubyear,$MONTH,$DAY, $HOUR,$MINUTE,$SECOND)
         = Timely::FullGregorian($NUMBR, 0,        0,        1);
                    # no local ^, no DST ^, is GPS ^
if(length $arg < 1)
  { # do nothing, accept defaults..
    my($pd) = Timely::PrettyDate(0,$stubyear,$MONTH,$DAY, $HOUR,$MINUTE,$SECOND);
    print "\n Testing default: " . substr($pd, 4);
    ShowEscMessage();
  }
elsif($arg =~ /^\s*(\d{4})\s+(\d{4})-(\d+)-(\d+)[ T]+(\d+):(\d+):(\d+)\s*$/ )
  { $myLowYear = int($1) - 1;   # low and high years
    $myTopYear = int($2);
    $MONTH  = $3;
    $DAY    = $4;
    $HOUR   = $5;
    $MINUTE = $6;
```

```
    $SECOND = $7;
  }
elsif($arg =~ /\s*(\d{4})-(\d+)-(\d+)[ T]+(\d+):(\d+):(\d+)\s*$/ ) # incl low year
  { $myLowYear = int($1) - 1;
    $MONTH  = $2;
    $DAY    = $3;
    $HOUR   = $4;
    $MINUTE = $5;
    $SECOND = $6;
  }
elsif($arg =~ /^\s*(\d+)-(\d+)[ T]+(\d+):(\d+):(\d+)\s*$/ ) # just the month..
  { $MONTH  = $1;
    $DAY    = $2;
    $HOUR   = $3;
    $MINUTE = $4;
    $SECOND = $5;
  }
elsif($arg =~ /^\s*(\d{4})\s+(\d{4})\s*$/ ) # a range of years
  { $myLowYear = int($1) - 1;
    $myTopYear = int($2);
  }
elsif($arg =~ /^\s*(\d{4})\s*$/ ) # just the minimum year
  { $myLowYear = int($1) - 1;
  }
 else
  { print "\n Unknown WALL format, expected MM-DD hh:mm:ss e.g. 12-31 20:30:00 ";
    return;
  };

if( ($myLowYear < $LOWYEAR)
  ||($myTopYear > $TOPYEAR-1)  # [check this]
  )
  { print "\n Bad year range $myLowYear .. $myTopYear ($LOWYEAR : " .($TOPYEAR-1). ")";
    return;
  };

  my($q) = "SELECT place, description FROM PLACES "
         . "WHERE p_amended between 2 and $TOPCOUNTRY AND reason > -1";
  # place is internal code, description is zone code:
  my(@ZDAT) = Timely::SQLManySQL($handDB, $q, 'get zones');
```

For each year, test the date for every zone. Simply determine the GPS time for that zone, based on the Gregorian date.

```
  my($YEAR) = $myTopYear;
  my($ERR) = 0;
  my($ANOMALIES) = 0;
  my($ref);

REDO: while($YEAR > $myLowYear)
```

```
{ print "\n$YEAR ";

if( CheckForEscape() )
  { $YEAR = $myLowYear-1;  # [hmm, check this]
    next REDO;             #
  };

foreach $ref(@ZDAT)
  { my($zid, $zone) = @$ref;
    my($JD, $shadow) = Timely::G2J(0, $zid,$YEAR,$MONTH,$DAY,$HOUR,$MINUTE,$SECOND);
  if($shadow < 0)  # if failed, don't try
    { my($yrs)
       = Timely::FetchTzDate($JD, $zone, $YEAR,$MONTH,$DAY,$HOUR,$MINUTE,$SECOND);
    if(length $yrs < 1)
      { Timely::XPrint(0, "\n  Bad time(" . int($shadow/60) . " min): "
        . Timely::PrettyDate(0,$YEAR, $MONTH, $DAY, $HOUR, $MINUTE, $SECOND)
        . " /$zone");
        $ANOMALIES ++;
      } else
      { Timely::XPrint(0, "\n   Mismatch! $zid/$zone NOT FOUND(sh=" .int($shadow/60)
          . ") v $yrs" . $TICK);
        $ERR++;
      };
    } else
    { # this is GPS time, so adjust to UTC, and then get back Gregorian:
      my($mine) = Timely::Greg($JD,0,0,1);  # date as YYYY-MM-DD hh:mm:ss
      $mine =~ s/ /T/;  # replace ' ' with 'T'
```

Next do the same thing using the tz database (in Perl).[4]

```
      # next do the same using the tz database:
      my($yours) =
        Timely::FetchTzDate($JD, $zone, $YEAR,$MONTH,$DAY,$HOUR,$MINUTE,$SECOND);
```

### 5.12.1 A TZ catch

There is a further catch — TZ skips the GMT value forward at the start of the shadow interval, while my code does this at the end. See Timely::InternalJulian (). Hence `$mine2` below:

```
    if($mine ne $yours)
      { my($mine2) = Timely::Greg($JD + $shadow, 0, 0, 1); # compensate for TZ jump!
        $mine2 =~ s/ /T/;
      if($mine2 eq $yours)
        { ## &Debug (0,"\n OK Zone $zone OK(+) $mine");
          print '+';
        } else
        { Timely::XPrint(0, "\n   Mismatch(TZ) $zid/$zone $mine v $yours" . $TICK);
```

[4] For use of eval/or do to trap an error in Perl, see e.g. http://stackoverflow.com/questions/10342875/how-to-properly-use-the-try-catch-in-perl-that-error-pm-provides.

```
            $ERR++;
          };
        } else
        { ## &Debug (0,"\n OK Zone $zone OK: $mine");
          print '.';
        };
      };
    };  # end foreach
    $YEAR --;
  };  # end while
```

Show error count and exit.

```
  Timely::XPrint( 0, "\n\n Errors=$ERR, $ANOMALIES anomalies "
    . ($myLowYear+1) . "--"
    . Timely::PrettyDate(0,$TOPYEAR, $MONTH, $DAY, $HOUR, $MINUTE, $SECOND) );
}
```

## 5.13 Around

Given a year, return all of the relevant transitions around that year (also for the preceding and following years), preformatted as date + zone offset + dst.

```
sub Around #
{ my($zone, $inp, $NUMBR);
    ($zone, $inp, $NUMBR)=@_;

  my($handDB) = $smalltime_h;

  my($YEARTOP);
  my($YEARBOT);
if( $inp =~ /^\s*(\d{4})\s+(\d{4})\s*$/ )
  { $YEARTOP = $2;
    $YEARBOT = $1;
  }
elsif($inp =~ /^\s*(\d{4}\s*$)/ ) #  YYYY
  { $YEARTOP = $1+1;       # defaults to year-1 .. year+1
    $YEARBOT = $1-1;
  }
elsif($inp =~ /^\s*$/ )  # just space or empty
  { my($YYYY,$MM,$DD,$h,$m,$s, undef,undef, $hog, undef, undef)
               # dummy values  ^    ^            ^      ^
     = Timely::J2G(1, $zone, $NUMBR);
  if(! $YYYY)
    { return(" Bad year for 'around' using current: "
             . &JulianDay($NUMBR) . "\n"); # unlikely?!
    };
    $YEARTOP = $YYYY+1;
    $YEARBOT = $YYYY-1;
```

```
  }
else
  { return( "Bad parameters for a '$inp'\n" );
  };

  # better to sort here than in database, but for now...
  my($q) = "SELECT transition, dst, zone_offset, ignored from timely "
    . "WHERE region = $zone AND year BETWEEN $YEARBOT AND $YEARTOP "
    . "ORDER BY transition";  # ASCending order
  ##  . "AND ignored = 0 ORDER BY transition";
      # ENCOMPASSES year of interest!
  my(@ROWS) = Timely::SQLManySQL($handDB, $q, 'get all rows');

  my($r);
  my(@P) = ();
foreach $r (@ROWS)
  { my($TR, $D, $Z, $ign) = @$r;  # transition, dst, zone, ignored
    my($flag) = '';
  if($ign)
    { $flag = '[x]'; # signal this is supplementary, can be 'ignored'
    };
    my($tr) = Timely::HugeToJ($TR);
    my($d)  = Timely::HugeToJ($D);
    my($z)  = Timely::HugeToJ($Z);
    push(@P, sprintf( "%.10f", &JulianDay($tr) ) . ' / ' . &Dat($tr)
         . ' z=' . &Tim($z) . ' d=' . &Tim($d) . ' --> '
         . &Dat( $tr+$z+$d ) . " $flag" );
    # Dat_() value is GPS time, not adjusted for zone/DST.
  };
  ## separable..

  print "\n Transitions for $zone ($inp); "
      . "[x] signifies 'Ignorable: end of year prosthetic':" ;
  my($rw);
foreach $rw (@P)
  { print "\n $rw";
  };
  print "\n";
  return(''); # ok
}
```

I append [x] to rows that have been inserted for convenience (at year end) but can be ignored.

#### 5.13.0.1 Selective logging

If `$BUG` is nonzero, then only do we log the line. We also allow the submitted parameter ($naah) to suppress logging.

```
sub Debug #
{ my($naah, $msg) = @_;
```

```
if(! $BUG)
  { return;
  };
if($naah)
  { return;
  };
  Timely::Log($msg);
}
```

## 5.14 List leap seconds

```
sub ListLeapseconds #
{ my($handDB) = $smalltime_h;

  my($q) = "SELECT toffset, utctime from leapseconds WHERE 1 ";
  my(@ROWS) = Timely::SQLManySQL($handDB, $q, 'get leapseconds');

  my($r);
  print "\n ================================================\n";
  print "                LIST OF LEAPSECONDS\n";
  print " ------------------------------------------------\n";
  print  " Offset  Julian timestamp       Gregorian date\n";
foreach $r (@ROWS)
  { my($offset, $TM) = @$r;
    my($sp) = '';
  if(length $offset < 2)
    { $sp = ' ';
    };
    my($J) = $TM/1000000; # Julian day number, NOT GPS adjusted
    my($YY, $MM, $DD, $h, $m, $s) = Timely::FullGregorian($J, 0, 0, 0); # $J in seconds
    print "  $sp $offset      " . sprintf( "%.4f", &JulianDay($J) )
        . "       " . Timely::PrettyDate(0,$YY,$MM,$DD,$h,$m,$s) . "\n";
  };
  print " ------------------------------------------------\n";
}
```

## 5.15 Test one zone

Accepts:

**zone** The zone

**inp** A text input of parameter(s)

**intrv** The interval between tests, in seconds.

Within a range of years (or just one), test the current, instantiated zone at a given interval. It's important to realise that we are simply testing each (incremented) Julian value by converting it to Gregorian using my internal algorithm, and back again. The Julian value increases monotonically, while we expect the Gregorian to jump around in accordance with

DST and zone changes. If the Gregorian to Julian routine (G2J) is working correctly, then jumps forward at the start of DST should be seamless, and jumps back should be evidenced by a "shadow" value equal to the magnitude of the "groundhog time" spent running the hour again (as the Julian day value increases inexorably).

```
sub TestOneZone #
{ my($zone, $inp, $intrv);
    ($zone, $inp, $intrv)=@_;
  # might here check/sanitise, ensure intrv >= 1;

  my($YYYY, $MM, $DD, $h, $m, $s,
     $eYYYY, $eMM, $eDD, $eh, $em, $es) = &ReadDateRange($inp);
if( $zone == Timely::GetUtcCode() )
  { print "Not tested, zone is UTC\n";
    return;
  };
```

Checks:

```
if( ($YYYY < 1)
  ||($eYYYY < 1)
  )
  { print "Bad argument for zone '$inp'\n";
    return;
  };
if($eYYYY < $YYYY)
  { print "Start $YYYY can't be before end $eYYYY\n";
    return;
  };

if($intrv < 1)
  { print "\n Interval must be a second or greater";
    return;
  };
  print " Testing zone $zone from "
      . Timely::PrettyDate(0, $YYYY, $MM, $DD, $h, $m, $s)
      . " to " . Timely::PrettyDate(0,  $eYYYY, $eMM, $eDD, $eh, $em, $es)
      . ", interval=$intrv" . "s\n";
  ShowEscMessage();
```

Setup. Note that `$shadow` values are in seconds.

```
  my($JHi, $shadow1) = Timely::G2J(1, $zone, $eYYYY, $eMM, $eDD, $eh, $em, $es); #
if( $JHi < MINIMUMDATE)
  { Timely::Warn(3, "Bad top value in '$inp' (code $shadow1)");
    return;
  };

  my($J, $shadow2) = Timely::G2J(1, $zone, $YYYY, $MM, $DD, $h, $m, $s); #
if($J < MINIMUMDATE)
```

```
  { print "\nBad start value in '$inp' (code $shadow2)";
    return;
  };    # might similarly check $shadow2

  my($oops) = 0;
  my($ok) = 0;
```

Loop around. From J2G () retrieve a Gregorian timestamp and then convert this back to Julian. If the two values are discordant, we have a problem—alert to this. See how G2J () also returns a "shadow" value. This is ordinarily zero, but may be positive or negative if we have an ambivalent wall time (Gregorian time), noting that when the zone or daylight saving falls back, one wall time will map to two Julian timestamps.

```
REDO: while($J < $JHi)
  {
  if( CheckForEscape() )
    { $J = $JHi;
      next REDO;             #
    };

    my($YY, $MM, $DD, $h, $m, $s,undef, undef, $hog, undef, undef)
                  # dummy values  ^    ^            ^      ^
          = Timely::J2G(1, $zone, $J);
    my($K, $shadow3) = Timely::G2J(1, $zone, $YY,$MM,$DD, $h,$m,$s);
    my($delta) = $K-$J;

  if( (abs($delta) > 1)
    || $shadow3
    )
    { my($d) = $delta; # for now simply transfer [explore]
    if( ( abs($d+$shadow3) > 0 )  # if shadow doesn't accommodate discrepancy
      &&($d)                      # and not just the first groundhog
      )
      { Timely::XPrint(0, "\n Discrepancy($d) at $J = "
        . Timely::PrettyDate(0,$YY,$MM,$DD,$h,$m,$s)
        . " (shadow=$shadow3), interval=$intrv, J="
        . sprintf("%.12f", $J) . ' K=' . sprintf("%.12f", $K) );
        $oops ++;
        print '?';
      if($oops > $MAXOOPS)
        { print "\n Too many errors!";
          return;
        };
      } else
      {
      if($d)
        { print '+';   # signal concordant shadow value
        } else
        { print '-';   # signal first shadow transition
```

```
        };
      };
    };
```

Pretty dots:

```
  if($ok % (50*$PRETTYDOTS) == 0) # every $PRETTYDOTS dots
    { print "\n" . &Dat($K) . ' ';
    };
  if($ok % 50 == 0)  # every 50
    { print '.';
    };
    $ok ++;
    $J += $intrv; # use this instead of increasing $J by a fraction of a day
  };
  Timely::XPrint(0, "\n Internal reconciliation for zone $zone. Errors=$oops/$ok\n");
}
```

### 5.15.1 Test and reconcile zone in range

Given zone, testing interval and range of years, test at that interval as follows:

1. Get UTC value for year start, and convert to Julian;
2. Convert:
   (a) Convert using Timely to the corresponding wall time;
   (b) Do the same using tz (DateTime);
   (c) Issue a warning if the two don't match
   (d) Increment the Julian value by the specified interval, and repeat.

This is similar to the 't' command in its use of timestamps, but performs an external reconciliation.

```
sub ReconcileOneZone #
{ my($zone, $zoneNAME, $inp, $intrv);
    ($zone, $zoneNAME, $inp, $intrv)=@_;

if( $zone == Timely::GetUtcCode() )
  { print "Not tested, zone is UTC\n";
    return;
  };
```

Initial checks:

```
 my($YYYY, $MM, $DD, $h, $m, $s,
   $eYYYY, $eMM, $eDD, $eh, $em, $es) = &ReadDateRange($inp);
if( ($YYYY < 1)
  ||($eYYYY < 1)
  )
  { print "Bad argument for zone '$inp'\n";
```

```
    return;
  };
if($eYYYY < $YYYY)
  { print "Start $YYYY can't be before end $eYYYY\n";
    return;
  };

if($intrv < 1)
  { print "\n Interval must be a second or greater";
    return;
  };
  print " Reconciling zone $zone from $YYYY-$MM-$DD $h:$m:$s to "
      . "$eYYYY-$eMM-$eDD $eh:$em:$es, interval=$intrv\n";
  ShowEscMessage();

  my($JHi) = Timely::GpsJulian($eYYYY, $eMM, $eDD, $eh, $em, $es,  0, 0, 0);
if( $JHi < MINIMUMDATE)
  { Timely::Warn(3, "$eYYYY is too small");
    return;
  };
  my($J) = Timely::GpsJulian($YYYY, $MM, $DD, $h, $m, $s,  0, 0, 0);
if($J < MINIMUMDATE)
  { Timely::Warn(3, "$YYYY is too small");
    return;
  };
```

Main loop:

```
  my($JMED) = $J; # seconds [fixup!]
  my($ERR) = 0;
  my($ANOMALIES) = 0; # [do we need this? explore]
  my($ok) = 0;
REDO: while($J < $JHi)
  {
  if( CheckForEscape() )
    { $J = $JHi;
      next REDO;             #
    };

    #1. Turn Julian timestamp into UTC Gregorian
    $J = $JMED; # prevent cumulative error
    my($YEAR, $jMM, $jDD, $jh, $jm, $js) = Timely::FullGregorian($J, 0, 0, 1);

    #2. Turn Julian into *local* zone Gregorian
    my($zYYYY,$zMM,$zDD,$zh,$zm,$zs, undef, undef, $hog, undef, undef)
                      # dummy values  ^    ^             ^      ^
       = Timely::J2G(1, $zone, $J);
```

Anomaly?

```
  if($zYYYY == 0)  # anomalous date
```

```
    { Timely::XPrint(0, "\n   'Anomaly' skipped: $zone $J");
      $ANOMALIES ++;
    } else
    { #3. Turn UTC Gregorian values into tz object
      my($dt1) = eval { DateTime->new( year => $YEAR, month => $jMM, day => $jDD,
         hour => $jh, minute => $jm, second => $js, time_zone => 'UTC') };

      #4. Turn tz into cloned object for this zone
      my($dtest) = eval { $dt1->clone->set_time_zone( $zoneNAME ) };
    if(! $dtest)
      { Timely::Warn(3, "INVALID(Perl DateTime test)$zone [$YEAR $jMM $jDD $jh $jm $js]
      } else
      { #5. compare local & tz
        my($mine) = Timely::PrettyDate(1,$zYYYY, $zMM, $zDD, $zh, $zm, $zs);
        my($yours) = $dtest->datetime();  # format is identical
      if($mine ne $yours)
        {
          Timely::XPrint( 0, "\n   Mismatch $mine v external $yours =>" . &Dat($J)
                   . ' <=' . sprintf("%.12f", $J) );
          $ERR++;
        };
      };
    };
```

Pretty up, continue:

```
  if($ok % (50*$PRETTYDOTS) == 0) # every $PRETTYDOTS dots
    { print "\n" . &Dat($J) . ' ';
    };
  if($ok % 50 == 0)  # every 50
    { print '.';
    };
    $ok ++;
    $JMED += $intrv; # use this instead  [fix me! no longer needed!!]
  };
```

Done:

```
  Timely::XPrint(0, "\n Reconciliation for zone $zone. "
           . "Errors=$ERR/$ok Anomalies=$ANOMALIES\n");
}
```

### 5.15.2 Read date range

Complex, allowing for one or two dates; if just one date, then the second defaults to the end of the first year. Either date can be in format YYYY or YYYY-MM-DD hh:mm:ss. Somewhat forgiving. Returns twelve (!) values, the start YYYY MM DD h m s and likewise for the end date. Does not validate the dates. On error, returns a first value of 0.

```
sub ReadDateRange #
```

```
{ my($inp);
    ($inp)=@_;

  my($YYYY, $MM, $DD, $h, $m, $s) = (0, 1, 1, 0, 0, 0);
  my($eYYYY, $eMM, $eDD, $eh, $em, $es) = (0, 12, 31, 23, 59, 59);

  my($more);
if($inp =~ /^\s*(\d{4})-(\d+)-(\d+)[ T](\d+):(\d+):(\d+)(.*)$/ )
  { ($YYYY, $MM, $DD, $h, $m, $s, $more) = ($1, $2, $3, $4, $5, $6, $7);
  if($more =~ /^\s*(\d{4})-(\d+)-(\d+)[ T](\d+):(\d+):(\d+)\s*$/ )
    { ($eYYYY, $eMM, $eDD, $eh, $em, $es) = ($1, $2, $3, $4, $5, $6);
    }
  elsif($more =~ /^\s*(\d{4})\s*$/ )
    { $eYYYY = $1;
    }
  elsif($more =~ /^\s*$/ )
    { $eYYYY = $YYYY;
    }
  else
    { # FAIL: leave $YYYY at zero!
    };
  }
elsif($inp =~ /^\s*(\d{4})\s+(\d{4})-(\d+)-(\d+)[ T](\d+):(\d+):(\d+)\s*$/ )
  { ($YYYY,  $eYYYY, $eMM, $eDD, $eh, $em, $es) = ($1,  $2, $3, $4, $5, $6, $7);
  }
elsif($inp =~ /^\s*(\d{4})\s+(\d{4})\s*$/ )
  { $YYYY = $1;
    $eYYYY = $2;
  }
elsif($inp =~ /^\s*(\d{4})\s*$/ )
  { $YYYY = $1;
    $eYYYY = $YYYY;
  }
else
  { # leave YYYY at 0.
  };
 return($YYYY, $MM, $DD, $h, $m, $s,  $eYYYY, $eMM, $eDD, $eh, $em, $es);
}
```

### 5.15.3 Show Escape Message

```
sub ShowEscMessage #
{
if($READKEYBLOCKINGPROBLEM)
  { return;  # do nothing
  };
  print " Press Esc to escape..\n";
}
```

### 5.15.4 Check for Escape

If Esc key has been pressed (and this feature is enabled) then return 1; otherwise 0. $READKEYBLOCKINGPROBLEM is a global used because, when I changed in Windows to ActivePerl under Perl v 5.28, I encountered an error where the console blocks on `if(defined($key))`, waiting for input, regardless of ReadMode specifications.

```
sub CheckForEscape
{
if($READKEYBLOCKINGPROBLEM)
  { return(0);
  };
  ReadMode 4; # unbuffered
  my($key) = ReadKey(-1);
  ReadMode 1; # normal.
if( defined($key) ) # this should not block
  {
  if(ord($key) == 27)  # Esc key pressed?
    { return(1);
    } else ## debug, but :
    { print "\n==$key==\n";
    };
  };
  return(0);
}
```

## 5.16 Dates

### 5.16.1 Julian tests

Do not confuse Julian day numbers with Julian dates.

When we store "Julian day numbers" in ***f*ehr**, we use big integer representations in microseconds since a reference point in the distant past (noon in Greenwich on January 1st, *minus 4712*); however it is more convenient (although less accurate) to use floating point calculations of *days* since this references time in both Perl and Javascript, as these default to floats. The Julian, floating point values are direct conversions of a familiar Gregorian date like "1901-01-01 12:00:00" to a corresponding float. But note there is also a *Julian calendar* last used in most countries in 1582. Even though we use Julian values, they refer to the (reformed) Gregorian calendar, so, for example, the above Gregorian date corresponds to 2415020.5 and not the matching Julian calendar date value of 2415032.5. (Take note that the Julian calendar took no account of fractional variations in leap values, simply allocating a leap day every fourth year; were you to stick to the Julian calendar, you'd have to accommodate this strangeness).

The following simply performs basic checks of the Julian conversion routines[5]:

[5]Reference at http://mysite.verizon.net/aesir_research/date/back.htm

```
sub TestJulian
{ Timely::XPrint(0, "\n *Testing reference dates* \n");
  my($fail) =  &TestPair('1901', 2415385.5, 1901,1,1,   0,0,0 );  # start 20th century
  $fail = &TestPair('UNIX', 2440587.5, 1970,1,1,   0,0,0) || $fail;  # UNIX reference
  $fail = &TestPair(' DOS', 2444239.5, 1980,1,1,   0,0,0) || $fail;  # DOS reference
  $fail = &TestPair('1900', 2415020.5, 1900,1,1,   0,0,0) || $fail;  # 1900

  $fail = &TestPair('GREG', 2299160.5, 1582,10,15, 0,0,0) || $fail;  # D1 Gregorian ref
  $fail = &TestPair('MJD0', 2400000.5, 1858,11,17, 0,0,0) || $fail;  # Modif. JD zero
  $fail = &TestPair('RAT1', 1721425.5, 1,1,1,      0,0,0) || $fail;  # Rata die = 1
  $fail = &TestPair('LAST', 2299149.5, 1582,10,4,  0,0,0) || $fail;  # 1D < Gregorian r

  $fail = &TestPair('1BC', 1721059.5, 0,1,1,       0,0,0) || $fail;  # D1, 1BC (year 0)
  $fail = &TestPair('JULI', 38,  -4712,1,1,       12,0,0) || $fail;  # JD reference
  Timely::XPrint(0, "\n\n");
if($fail)
  { ## die "\n*ERROR* Julian conversion(s) failed\n";
    return 1; #
  };

  print "You shouldn't need to press a key now: yet you may ... \n";
  my($key);
if(  defined( $key = ReadKey(-1) )  ) # this should not block
  { print "**Your Perl has a READ BLOCKING problem (accommodated)**\n\n";
    $READKEYBLOCKINGPROBLEM = 1;
  };
  return($fail);
}
```

The subsidiary TestPair (). Because the reference dates supplied in `$Jref` are Julian day numbers, we convert to seconds internally. A numeric value is returned—1 for failure, 0 for success.

```
sub TestPair #
{ my($name, $Jref, @SMART) = @_;
  my($fail) = 0;
  my($Zoff) = 0;
  my($DST)  = 0;
  my($delta) = 0.01; # 10ms. [explore]

  $Jref *= 86400; # convert to seconds

  my($J1) = Timely::ToJulian(@SMART, 0, $Zoff, $DST);
  my($diff) = $J1-$Jref;
if( abs($diff) > $delta)
  { $fail = 1;
    Timely::XPrint(0,  " -Failed- $name Julian test ($J1, d=$diff)\n" );
  };
  my(@GREG) = Timely::FullGregorian($J1,$Zoff,$DST,0);
    # no GPS adjust so $DST is a redundant variable [fix me]
```

```
if(! &SameArray(\@GREG, \@SMART))  # avoid smart match.
  { $fail = 1;
    Timely::XPrint(0,  " -Failed- $name Gregorian retest <@GREG>\n" );
  };

if(! $fail)
  { Timely::XPrint(0, $name . $TICK . ' ');  # eye candy
  };
  return($fail);
}
```

Check for identical arrays without smartmatch. It's sad that if you get rid of the 'experimental' @A ~~ @B, you come up with something like:

```
sub SameArray #
{ my($rA, $rB) = @_; # two array references

  my(@A) = @$rA;
  my(@B) = @$rB;
  my($aL) = scalar @A;
if($aL != scalar @B)
  { return(0);
  };
while($aL > 0)
  { $aL --;
  if($A[$aL] !~ /\Q$B[$aL]\E/) #note quotemeta
    { return(0);
    };
  };
  return(1);
}
```

## 5.17 Multitest

Fetch all transition points from **timely** where ignored is 0 (not just end-of-year timestamps) and check them for that timezone.

1. SELECT region, transition FROM timely WHERE ignored = 0;
2. For each of these:
   (a) conver transition from microseconds to a Julian day number using HugeToJ ()
   (b) convert to Gregorian timestamp in our program using J2G ()
   (c) Similarly convert to tz timestamp;
3. Compare the two, signal mismatch.

Have the option of constraining it to the current region; also v+n or v-n will add or subtract the specified number of seconds to each timestamp.

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

  my($handDB) = $smalltime_h;

  my($q) = "SELECT region, transition, description FROM timely t "
         . "INNER JOIN PLACES P on t.region = P.place WHERE ignored = 0 "
         . "AND t.year < $TOPYEAR" ;  ## limit top, prevent failure!
  my($all) = 0;

  # check v+ means add 1 second, v- subtracts
  my($OFFSET) = 0;
if($inp =~ /^\+(\d+)(.*)/)
  { $OFFSET = 1000000*$1; # 1 second=1M
    $inp = $2; # trim
  }
elsif($inp =~ /^(-\d+)(.*)/) # [ugly]
  { $OFFSET = 1000000*$1; # e.g. -1 s
    $inp = $2;
  };

if( $inp !~ /^\s*all\s*$/ ) # v all for the lot!
  { $q .= " and region = $region";
  } else
  { $all = 1;
  };

  my(@ZTRAN) = Timely::SQLManySQL($handDB, $q, 'get zone transitions');
  my($ERR) = 0;
  my($ANOMALIES) = 0;

  my($ref);
foreach $ref(@ZTRAN)
  { my($zid, $TR, $rName) = @$ref;

    my($JD) = Timely::HugeToJ($TR+$OFFSET);
    my($YYYY,$MM,$DD,$h,$m,$s, $dz, $dst, $hog, $deltz, $delds)
       = Timely::J2G(0, $zid, $JD); # JD is Gps-based, $dst is DST, and
       # $hog is zero unless we're in the groundhog hour.
  if($YYYY == 0)  # anomalous date
    { Timely::XPrint( 0, "\n     'Out of range' date skipped: $zid/$rName "
              . &JulianDay($JD) );
      $ANOMALIES ++;
    } else
    { my($mine) = Timely::Greg($JD,0,0,1);  # standard date, then in TZ!
      $mine =~ s/ /T/;  # replace ' ' with 'T'
      my($yours) =  Timely::FetchTzDate($JD, $rName, $YYYY, $MM, $DD, $h, $m, $s);
```

If there's a mismatch, first check whether we're in the 'groundhog'

zone ($hog is non-zero). A simple example would be where DST has gone from 1 hour to zero. There are then two Julian times that correspond to a single wall time. Let's say the transition is at midnight: then each wall time in the range 11:00:00 to midnight (not including midnight) will have two Julian values. At the precise moment that we reach midnight the first time around, we jump back, so this Julian and that one hour earlier will correspond. The earlier time is signalled by $hog = -1, the later by 1.

There is a subtlety here. In translating Julian to Gregorian, I choose the earlier wall time; but where tz translates a wall time to UTC, as is performed by FetchTzDate (), the later of the two timestamps is returned. We need to check for this discordance, and adjust.

```
if($mine ne $yours)
  {
  if($hog)  # $hog != 0
    { # crude hack [explore simply using delta values]

      # assume that: A. there are two dates
      #              B. My program returned the earlier one
      #              C. tz returned the later one
      #              D. the discrepancy is ($deltz + $delds)
      # E.g. Zone went back 2 h from +3 to +1 and DST went forward by 1 h,
      #   then deltz = -2 and delds = 1, effectively back -1 hour.
      #   Advance by the NEGATIVE of this value (tz takes the second time)

      my($truedelta) = $hog * ($deltz + $delds); # as is shadow, will be -ve.
      my($fixmine) = Timely::Greg($JD, $truedelta, 0, 1); # adjust "as if" zone
      $fixmine =~ s/ /T/;
    if($fixmine ne $yours)
      { $ERR ++;
        Timely::XPrint( 0, "\nBad transition $rName(" . &JulianDay($JD)
                  . ",$hog), mine=$mine tz=$yours, dz=" . &Tim($deltz)
                  . " dd=" . &Tim($delds) . ", failed fix=$fixmine, Z="
                  . &Tim($dz) . ", DST=" . &Tim($dst) );
        ##Timely::XPrint(0, "\n Transition Mismatch [ $mine | $yours, $rName($hog)
        ##           . &Tim($deltz) . ' ' . &Tim($delds) . " } ] "); # HMMM ???
      };
      #
      print &Sign($hog); # + or -
```

Otherwise, handle error:

```
    } else
    { $ERR ++;
    if(length $yours < 1)
      { $yours = '?';
      } else
      { Timely::XPrint(0, "\n");
      };
      Timely::XPrint( 0, "Mismatch for $rName($JD): mine=$mine, tz=$yours ("
               . Timely::PrettyDate(0,$YYYY,$MM,$DD,$h,$m,$s)
```

```
                    . ') z=' . &Tim($dz) . ' d=' . &Tim($dst) . ' ' );
        };
```

Repeat, end off:

```
      } else
      { print '.';
      };
    };
  };
  Timely::XPrint(0, "Errors=$ERR, $ANOMALIES anomalies\n\n");
}
```

## 5.18 Utilities

### 5.18.1 Exit

The supplied code value can be *text*, although the default is 0. When we exit, however, we return that 0, or if non-zero, a failure code of 2. We must only COMMIT if the supplied code is zero.

```
sub ImplementExit #
{ my($code) = @_;

if($code)
  { $code = 2; # if non-zero [nasty, $code can be text [hmm] explore]
  } else
  { my($qc) = "COMMIT";
    my($fail) = Timely::DoSQL($DATABASES{$BASEDATABASE}, $qc, 'commit');
    &Print(0, "\n  Result of commit: $fail");
  if($fail)
    { $code = $fail;
    };
  };

  my(@WARNS) = Timely::ListWarnings();
  Timely::XPrint(0, "\nWarning counts: \n -\t*\t**\t***\n " . join("\t",@WARNS) );

  my($db);
foreach $db (keys(%DATABASES))
  { DoClose( $db );
    &Print(2, "\n  Closed $db");
  };
  print "\n Goodbye($code). \n";

  Timely::EndTimely();

  exit($code);
}
```

#### 5.18.1.1 Actually close database

```
sub DoClose #
{ my($dbname);
    ($dbname)=@_;

  $DATABASES{$dbname}->disconnect();

  delete($DATABASES{$dbname}); # remove associative entry
  &Print(2, " : ODBC, Closed $dbname");
}
```

### 5.18.2 Sign

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

### 5.18.3 DoWarn

```
sub DoWarn #
{ my($level, $msg) = @_;
  Timely::Warn($level, $msg);
}
```

### 5.18.4 Print

```
sub Print #
{ my($level, $msg) = @_;
  Timely::XPrint($level, $msg);
}
```

### 5.18.5 Aagh (modified)

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

#### 5.18.5.1 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].

**Clear the count**

```
sub ClearNewWarnings #
{ $LOCALS{'wArnings'} = Timely::GetNewWarnings();
  Timely::ClearWarnings();
}
```

## 5.19 Pre-processing

We allow command-line args. The options that we're interested in are along these lines:

```
perl seek.pl user=2001;
```

The terminal semicolon after arguments is optional. The options relevant to small_time are listed in Section 5.19.1.

```
sub PreArgs
{
foreach my $arg (@ARGV)
  { ParseOneArg($arg);
  };
}
```

### 5.19.1 Parse single pre-arg

The current options are:

**user**= (Specify numeric identity of user cf. $USERID);

**db**= (Identity of database e.g. fehr; currently defaults to SMALLTIME)

**leapseconds**= (Current number of leapseconds);

**topyear**= alter $TOPYEAR, should not be under current year (Hmm, use with great caution).

–**help**

In this implementation, unlike in **vector seekwell**, we don't specify a script (sos=) nor do we allow a script line at the start. We allow some flexibility here—a comma can separate arguments, which can be in any order. Note that my original code allowed semicolons (in Windows) but these have a special meaning on the Linux command line, hence the current use of commas!

```
sub ParseOneArg #
{ my ($arg) = @_;

  # Extract specific pairs, first database:
if($arg =~ s/db=(\w+),?//)
{  $BASEDATABASE = $1;
   # print "\nDebug: TARGET DATABASE set to $BASEDATABASE\n";
}

if($arg =~ s/user=(\d+),?//) # Extract User ID
  { $USERID = $1;
    # print "\nDebug: User set to $USERID\n";
  }
```

```
if($arg =~ s/leapseconds=(\d+),?//)   # Extract Leapseconds
  { $LEAPSECONDS = $1;
    # print "\nDebug: LEAPSECOND COUNT = $LEAPSECONDS\n";
  }

if($arg =~ s/topyear=(\d{4}),?//)   # Extract year maximum
  { $TOPYEAR = $1;
  if($TOPYEAR < 2026)
    { exit(BadTopYear);
    };
    # print "\nDebug: TOP YEAR = $TOPYEAR\n";
  }

if($arg =~ /--help/)
  { print "\nCommand-line examples:\n user=2000 db=fehr leapseconds=27"
        . "\n";
    exit(OnlyHelp); #
  }

if($arg =~ /^\s*$/) # nothing of consequence
  { return;
  };

  print("Unused residual command-line argument: '$arg'\n");
}
```

This is fairly flexible, allowing terminal semicolons or even omission of spaces between; but ideally we should simply put in spaces and omit semicolons.

# 6.0 Batch files

These require the usual ***smalltime*** installation. In LyX, say File | Export | LaTeX(XeTeX) for both timely_400.lyx and small_time_400.lyx.

The first time, within the *~/smalltime/* directory ensure that *Dogwagger405.pl* is present, and say:

**`perl ./Dogwagger405.pl small_time_400.tex`**

Subsequently, the following batch files should work.

## 6.1 makesmall.sh

Once created, run this by saying **`bash makesmall.sh`** from the Linux command line. For Windows, see later.

```
  perl ./Dogwagger405.pl timely_400.tex
if [ $? -ne 0 ]
then
  echo Script aborted.
  exit 1
fi
  perl ./Dogwagger405.pl small_time_400.tex
```

The `if [ $? -ne ...` instruction tests for the error code returned by the Perl script if it fails.

## 6.2 makesmall.bat

This extracts files from the current .tex file. Run from the Windows command line by simply saying **`make`** :

```
    echo off
    cls
    perl Dogwagger405.pl timely_400.tex || (goto :FAILED)
    perl Dogwagger405.pl small_time_400.tex || (goto :FAILED)
    echo Success!
    exit /b 0
:FAILED
    echo Error level was %errorlevel%
    exit /b %errorlevel%
```

(Note that because we overwrite the file while it's executing as a batch, you might occasionally encounter some anomalies).

## 6.3 runsmall.sh

This assumes we're in ~/smalltime/ and a copy of Dogwagger405.pl is present in this directory. There say:

```
bash runsmall.sh
```

The following could be fancied up a bit.

```
cd perl
perl small.pl
cd ..
```

## 6.4 runsmall.bat

```
cls
cd perl
perl small.pl
cd ..
rem that's it
```

■

# A.0 Problems & To Do List

## A.1 Gotchas

NB. If your Perl date/time is not up to date with the current IANA tz rules, then discrepancies will abound.

1. Currently we need DELETE option permitted for the ODBC-associated user (e.g. ‘vanilla’). Otherwise perl may crash.

2. If the initial running of `perl small.pl` fails, then we may be left with residual rows in some tables, as currently the failure isn’t transmitted to the caller, and the commit still seems to happen. If this happens, then manually enter mariadb/mysql and delete everything in tz_cutoffs, tz_rules, timely and leapseconds. ALSO SAY:

   ```
   DELETE FROM PLACES WHERE place BETWEEN 1 AND 999;
   ```

   Caution is clearly advised.

# B.0 Amendment Log

## B.1 Version 1.0

This is the base version.

## B.2 Version 4.0

A sudden jump to synch with timely 4.0 (Okay, I know). The main idea here is a substantial revision, hoiking all of the user-interface-based stuff out of the timely Perl module—it didn't really belong there in a module.

1. Write an into that describes:
   (a) The absolute minimum files (timely, PLACES) with contents (put these up on GitHub too)
   (b) How to move these from **smalltime** to another database such as ***f*ehr**, honouring place identities.
2. Write *runsmall.sh*
3. Must still test:
   (a) Check using: `v all`; Initially had problems with new zone: America/Coyhaique [explore] This was because the Perl version was not completely up to date:

       ```
       The timezone 'America/Coyhaique' could not be loaded,
       or is an invalid name
       ```

       In other words, our tz version is more recent than the Perl "reference" one!
       [The 'problem' disappears when the Perl is updated by: `sudo cpanm DateTime::TimeZone` ]
4. Need to support cmd line parameters NB. database name [ok]