```c
#include <ruby.h>

#ifndef DBL2NUM
#define DBL2NUM(dbl) rb_float_new(dbl)
#endif

#define R2D 57.295779513082320876798154814105
#define D2R 0.017453292519943295769236907684886
#define M2PI M_PI * 2.0

static VALUE
t_init(VALUE self)
{
  return self;
}

static VALUE
func_mean_anomally(VALUE self, VALUE vd)
{
  double vma = fmod((357.5291  + 0.98560028 * NUM2DBL(vd)) * D2R, M2PI);
  return DBL2NUM(vma);
}

static VALUE
func_eccentricity(VALUE self, VALUE vd)
{
  double ve = 0.016709 - 1.151e-9 * NUM2DBL(vd);
  return DBL2NUM(ve);
}

static VALUE
func_equation_of_center(VALUE self, VALUE vd)
{
  double vma = NUM2DBL(func_mean_anomally(self, vd));
  double ve = NUM2DBL(func_eccentricity(self, vd));
  double vesqr = ve * ve;
  double vecube = ve * ve * ve;
  double veoc = 2.0 * ve * sin(vma) +
              5.0 / 4.0 * vesqr * sin(2 * vma) +
              vecube / 12.0 * (13.0 * sin(3 * vma) - 3.0 * sin(vma));
  return DBL2NUM(veoc);
}

static VALUE
```

```
func_true_anomally(VALUE self, VALUE vd)
{
  double vma = NUM2DBL(func_mean_anomally(self, vd));
  double veoc = NUM2DBL(func_equation_of_center(self, vd));
  double vta = vma + veoc;
  return DBL2NUM(vta);
}

static VALUE
func_mean_longitude(VALUE self, VALUE vd)
{
  double vml = fmod(280.4664567 * D2R +
                      0.9856473601037645 * D2R * NUM2DBL(vd), M2PI);
  return DBL2NUM(vml);
}

static VALUE
func_eccentric_anomally(VALUE self, VALUE vd)
{
  double ve = NUM2DBL(func_eccentricity(self, vd));
  double vml = NUM2DBL(func_mean_longitude(self, vd));
  double vea = vml + ve * sin(vml) * (1.0 + ve * cos(vml));
  return DBL2NUM(vea);
}

static VALUE
func_obliquity_of_ecliptic(VALUE self, VALUE vd)
{
  double vooe = (23.439291 - 3.563E-7 * NUM2DBL(vd)) * D2R;
  return DBL2NUM(vooe);
}

static VALUE
func_longitude_of_perihelion(VALUE self, VALUE vd)
{
  double vlop = fmod(282.9404 * D2R + 4.70935e-05 * D2R * NUM2DBL(vd), M2PI);
  return DBL2NUM(vlop);
}

static VALUE
func_xv(VALUE self, VALUE vd)
{
  double vea = NUM2DBL(func_eccentric_anomally(self, vd));
  double ve = NUM2DBL(func_eccentricity(self, vd));
  double vxv = cos(vea) - ve;
```

```
  return DBL2NUM(vxv);
}

static VALUE
func_yv(VALUE self, VALUE vd)
{
  double vea = NUM2DBL(func_eccentric_anomally(self, vd));
  double ve = NUM2DBL(func_eccentricity(self, vd));
  double vyv = sqrt(1.0 - ve * ve) * sin(vea);
  return DBL2NUM(vyv);
}

static VALUE
func_true_longitude(VALUE self, VALUE vd)
{
  double vta = NUM2DBL(func_true_anomally(self, vd));
  double vlop = NUM2DBL(func_longitude_of_perihelion(self, vd));
  double vtl = fmod(vta + vlop, M2PI);
  return DBL2NUM(vtl);
}

static VALUE
func_rv(VALUE self, VALUE vd)
{
  double vxv = NUM2DBL(func_xv(self, vd));
  double vyv = NUM2DBL(func_yv(self, vd));
  double vrv = sqrt(vxv * vxv + vyv * vyv);
  return DBL2NUM(vrv);
}

static VALUE
func_ecliptic_x(VALUE self, VALUE vd)
{
  double vrv = NUM2DBL(func_rv(self, vd));
  double vtl = NUM2DBL(func_true_longitude(self, vd));
  double vex = vrv * cos(vtl);
  return DBL2NUM(vex);
}

static VALUE
func_ecliptic_y(VALUE self, VALUE vd)
{
  double vrv = NUM2DBL(func_rv(self, vd));
  double vtl = NUM2DBL(func_true_longitude(self, vd));
  double vey = vrv * sin(vtl);
```

```
  return DBL2NUM(vey);
}

static VALUE
func_right_ascension(VALUE self, VALUE vd)
{
  double vey = NUM2DBL(func_ecliptic_y(self, vd));
  double vooe = NUM2DBL(func_obliquity_of_ecliptic(self, vd));
  double vex = NUM2DBL(func_ecliptic_x(self, vd));
  double vra = fmod(atan2(vey * cos(vooe), vex) + M2PI, M2PI);
  return DBL2NUM(vra);
}

static VALUE
func_declination(VALUE self, VALUE vd)
{
  double vex = NUM2DBL(func_ecliptic_x(self, vd));
  double vey = NUM2DBL(func_ecliptic_y(self, vd));
  double vooe = NUM2DBL(func_obliquity_of_ecliptic(self, vd));
  double ver = sqrt(vex * vex + vey * vey);
  double vz = vey * sin(vooe);
  double vdec = atan2(vz, ver);
  return DBL2NUM(vdec);
}

static VALUE
func_sidetime(VALUE self, VALUE vd)
{
  double vst = fmod((180 + 357.52911 + 282.9404) +
    (0.985600281725 + 4.70935E-5) * NUM2DBL(vd), 360.0);
  return DBL2NUM(vst);
}

static VALUE
func_dlt(VALUE self, VALUE vd, VALUE vlat)
{
  double vsin_alt = sin(-0.8333 * D2R);
  double vlat_r = NUM2DBL(vlat) * D2R;
  double vcos_lat = cos(vlat_r);
  double vsin_lat = sin(vlat_r);
  double vooe = NUM2DBL(func_obliquity_of_ecliptic(self, vd));
  double vtl = NUM2DBL(func_true_longitude(self, vd));
  double vsin_dec = sin(vooe) * sin(vtl);
  double vcos_dec = sqrt( 1.0 - vsin_dec * vsin_dec );
  double vdl = acos((vsin_alt - vsin_dec * vsin_lat) / (vcos_dec * vcos_lat));
```

```
  double vdla = vdl * R2D;
  double vdlt = vdla / 15.0 * 2.0;
  return DBL2NUM(vdlt);
}

void Init_calc_sun(void)
{
  VALUE cCalcSun = rb_define_class("CalcSun", rb_cObject);
  rb_define_method(cCalcSun, "initialize", t_init, 0);
  rb_define_method(cCalcSun, "mean_anomally", func_mean_anomally, 1);
  rb_define_method(cCalcSun, "eccentricity", func_eccentricity, 1);
  rb_define_method(cCalcSun, "equation_of_center", func_equation_of_center, 1);
  rb_define_method(cCalcSun, "true_anomally", func_true_anomally, 1);
  rb_define_method(cCalcSun, "mean_longitude", func_mean_longitude, 1);
  rb_define_method(cCalcSun, "eccentric_anomally", func_eccentric_anomally, 1);
  rb_define_method(cCalcSun, "obliquity_of_ecliptic", func_obliquity_of_ecliptic,
1);
  rb_define_method(cCalcSun, "longitude_of_perihelion",
func_longitude_of_perihelion, 1);
  rb_define_method(cCalcSun, "xv", func_xv, 1);
  rb_define_method(cCalcSun, "yv", func_yv, 1);
  rb_define_method(cCalcSun, "true_longitude", func_true_longitude, 1);
  rb_define_method(cCalcSun, "rv", func_rv, 1);
  rb_define_method(cCalcSun, "ecliptic_x", func_ecliptic_x, 1);
  rb_define_method(cCalcSun, "ecliptic_y", func_ecliptic_y, 1);
  rb_define_method(cCalcSun, "right_ascension", func_right_ascension, 1);
  rb_define_method(cCalcSun, "declination", func_declination, 1);
  rb_define_method(cCalcSun, "sidereal_time", func_sidetime, 1);
  rb_define_method(cCalcSun, "dlt", func_dlt, 2);
}
```

```
lib = File.expand_path('../lib', __FILE__)
$LOAD_PATH.unshift(lib) unless $LOAD_PATH.include?(lib)
require 'calc_sun'

require 'date'
J2000 = DateTime.parse('2000-01-01T12:00:00').jd
INV360 = 1.0 / 360.0
def rev180(x)
  x - 360.0 * (x * INV360 + 0.5).floor
end

include Math
lat = 41.95
```

```
lon = -88.75
day = Date.parse('2016-11-29')
jd = day.jd
d = jd - lon / 360.0 - J2000
cs = CalcSun.new
st = cs.sidereal_time(d)
lst = (st + 180 + lon) % 360.0

ra = cs.right_ascension(d) * 180 / PI
t_south = 12.0 - rev180(lst - ra) / 15.0

diurnal_arc = cs.dlt(d, lat) / 2.0
rise = t_south - diurnal_arc
set = t_south + diurnal_arc

printf("\n")

printf("\tSun rises \t\t\t : %2.0f:%02.0f UTC\n",
       rise.floor, (rise % 1 * 60.0).floor)

printf("\tSun at south \t\t : %2.0f:%02.0f UTC\n",
       ((rise + set) / 2.0).floor,
       (((rise + set) / 2.0 % 1.0) * 60).floor)

printf("\tSun sets \t\t\t : %2.0f:%02.0f UTC\n",
       set.floor, (set % 1 * 60.0).floor)
```