```
                        // valid range: -12   to   12 hours,   error code: 8
    double longitude;   // Observer longitude (negative west of Greenwich)
                        // valid range: -180  to  180 degrees, error code: 9
    double latitude;    // Observer latitude (negative south of equator)
                        // valid range: -90   to   90 degrees, error code: 10
    double elevation;   // Observer elevation [meters]
                        // valid range: -6500000 or higher meters,    error code: 11
    double pressure;    // Annual average local pressure [millibars]
                        // valid range:    0 to 5000 millibars,       error code: 12
    double temperature; // Annual average local temperature [degrees Celsius]
                        // valid range: -273 to 6000 degrees Celsius, error code; 13
    double slope;       // Surface slope (measured from the horizontal plane)
                        // valid range: -360 to 360 degrees, error code: 14
    double azm_rotation; // Surface azimuth rotation (measured from south to projection of
                        //     surface normal on horizontal plane, negative west)
                        // valid range: -360 to 360 degrees, error code: 15
    double atmos_refract;// Atmospheric refraction at sunrise and sunset (0.5667 deg is typical)
                        // valid range: -5   to   5 degrees, error code: 16
    int function;       // Switch to choose functions for desired output (from enumeration)

    //-----------------Intermediate OUTPUT VALUES--------------------

    double jd;          //Julian day
    double jc;          //Julian century
    double jde;         //Julian ephemeris day
    double jce;         //Julian ephemeris century
    double jme;         //Julian ephemeris millennium
    double l;           //earth heliocentric longitude [degrees]
    double b;           //earth heliocentric latitude [degrees]
    double r;           //earth radius vector [Astronomical Units, AU]
    double theta;       //geocentric longitude [degrees]
    double beta;        //geocentric latitude [degrees]
    double x0;          //mean elongation (moon-sun) [degrees]
    double x1;          //mean anomaly (sun) [degrees]
    double x2;          //mean anomaly (moon) [degrees]
    double x3;          //argument latitude (moon) [degrees]
    double x4;          //ascending longitude (moon) [degrees]
    double del_psi;     //nutation longitude [degrees]
    double del_epsilon; //nutation obliquity [degrees]
    double epsilon0;    //ecliptic mean obliquity [arc seconds]
    double epsilon;     //ecliptic true obliquity  [degrees]
    double del_tau;     //aberration correction [degrees]
    double lamda;       //apparent sun longitude [degrees]
    double nu0;         //Greenwich mean sidereal time [degrees]
    double nu;          //Greenwich sidereal time [degrees]
    double alpha;       //geocentric sun right ascension [degrees]
    double delta;       //geocentric sun declination [degrees]
    double h;           //observer hour angle [degrees]
    double xi;          //sun equatorial horizontal parallax [degrees]
    double del_alpha;   //sun right ascension parallax [degrees]
    double delta_prime; //topocentric sun declination [degrees]
    double alpha_prime; //topocentric sun right ascension [degrees]
    double h_prime;     //topocentric local hour angle [degrees]
    double e0;          //topocentric elevation angle (uncorrected) [degrees]
    double del_e;       //atmospheric refraction correction [degrees]
    double e;           //topocentric elevation angle (corrected) [degrees]
    double eot;         //equation of time [minutes]
    double srha;        //sunrise hour angle [degrees]
    double ssha;        //sunset hour angle [degrees]
    double sta;         //sun transit altitude [degrees]

    //---------------------Final OUTPUT VALUES------------------------

    double zenith;      //topocentric zenith angle [degrees]
    double azimuth180;  //topocentric azimuth angle (westward from south) [-180 to 180 degrees]
    double azimuth;     //topocentric azimuth angle (eastward from north) [   0 to 360 degrees]
    double incidence;   //surface incidence angle [degrees]
    double suntransit;  //local sun transit time (or solar noon) [fractional hour]
    double sunrise;     //local sunrise time (+/- 30 seconds) [fractional hour]
    double sunset;      //local sunset time (+/- 30 seconds) [fractional hour]
} spa_data;

//Calculate SPA output values (in structure) based on input values passed in structure
int spa_calculate(spa_data *spa);

#endif
```

A.7. C Code: SPA source file (SPA.c)

```
//////////////////////////////////////////
//      Solar Position Algorithm (SPA)     //
//                   for                   //
//        Solar Radiation Application      //
//                                         //
//               May 12, 2003              //
//                                         //
//   Filename: SPA.C                       //
//                                         //
//   Afshin Michael Andreas                //
//   afshin_andreas@nrel.gov (303)384-6383 //
//                                         //
//   Measurement & Instrumentation Team    //
//   Solar Radiation Research Laboratory   //
//   National Renewable Energy Laboratory  //
//   1617 Cole Blvd, Golden, CO 80401      //
//////////////////////////////////////////

//////////////////////////////////////////
//   See the SPA.H header file for usage   //
//                                         //
//   This code is based on the NREL        //
//   technical report "Solar Position      //
//   Algorithm for Solar Radiation         //
//   Application" by I. Reda & A. Andreas  //
//////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////
//
//   NOTICE
//
//This solar position algorithm for solar radiation applications (the "data") was produced by
//the National Renewable Energy Laboratory ("NREL"), which is operated by the Midwest Research
//Institute ("MRI") under Contract No. DE-AC36-99-GO10337 with the U.S. Department of Energy
//(the "Government").
//
//Reference herein, directly or indirectly to any specific commercial product, process, or
//service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply
//its endorsement, recommendation, or favoring by the Government, MRI or NREL.
//
//THESE DATA ARE PROVIDED "AS IS" AND NEITHER THE GOVERNMENT, MRI, NREL NOR ANY OF THEIR
//EMPLOYEES, MAKES ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING THE WARRANTIES OF MERCHANTABILITY
//AND FITNESS FOR A PARTICULAR PURPOSE, OR ASSUMES ANY LEGAL LIABILITY OR RESPONSIBILITY FOR THE
//ACCURACY, COMPLETENESS, OR USEFULNESS OF ANY SUCH INFORMATION DISCLOSED IN THE ALGORITHM, OR
//OF ANY APPARATUS, PRODUCT, OR PROCESS DISCLOSED, OR REPRESENTS THAT ITS USE WOULD NOT INFRINGE
//PRIVATELY OWNED RIGHTS.
//
////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////
// Revised 27-FEB-2004 Andreas
//         Added bounds check on inputs and return value for spa_calculate().
// Revised 10-MAY-2004 Andreas
//         Changed temperature bound check minimum from -273.15 to -273 degrees C.
// Revised 17-JUN-2004 Andreas
//         Corrected a problem that caused a bogus sunrise/set/transit on the equinox.
// Revised 18-JUN-2004 Andreas
//         Added a "function" input variable that allows the selecting of desired outputs.
// Revised 21-JUN-2004 Andreas
//         Added 3 new intermediate output values to SPA structure (srha, ssha, & sta).
// Revised 23-JUN-2004 Andreas
//         Enumerations for "function" were renamed and 2 were added.
//         Prevented bound checks on inputs that are not used (based on function).
// Revised 01-SEP-2004 Andreas
//         Changed a local variable from integer to double.
// Revised 12-JUL-2005 Andreas
//         Put a limit on the EOT calculation, so that the result is between -20 and 20.
// Revised 26-OCT-2005 Andreas
//         Set the atmos. refraction correction to zero, when sun is below horizon.
//         Made atmos_refract input a requirement for all "functions".
//         Changed atmos_refract bound check from +/- 10 to +/- 5 degrees.
// Revised 07-NOV-2006 Andreas
//         Corrected 3 earth periodic terms in the L_TERMS array.
//         Corrected 2 earth periodic terms in the R_TERMS array.
```

```
// Revised 10-NOV-2006 Andreas
//         Corrected a constant used to calculate topocentric sun declination.
//         Put a limit on observer hour angle, so result is between 0 and 360.
// Revised 13-NOV-2006 Andreas
//         Corrected calculation of topocentric sun declination.
//         Converted all floating point inputs in spa structure to doubles.
// Revised 27-FEB-2007 Andreas
//         Minor correction made as to when atmos. refraction correction is set to zero.
//////////////////////////////////////////////////////////////////////////////////////////

#include <math.h>
#include "spa.h"

#define PI         3.14159265358979323846264338327950288419 71
#define SUN_RADIUS 0.26667

#define L_COUNT 6
#define B_COUNT 2
#define R_COUNT 5
#define Y_COUNT 63

#define L_MAX_SUBCOUNT 64
#define B_MAX_SUBCOUNT 5
#define R_MAX_SUBCOUNT 40

enum {TERM_A, TERM_B, TERM_C, TERM_COUNT};
enum {TERM_X0, TERM_X1, TERM_X2, TERM_X3, TERM_X4, TERM_X_COUNT};
enum {TERM_PSI_A, TERM_PSI_B, TERM_EPS_C, TERM_EPS_D, TERM_PE_COUNT};
enum {JD_MINUS, JD_ZERO, JD_PLUS, JD_COUNT};
enum {SUN_TRANSIT, SUN_RISE, SUN_SET, SUN_COUNT};

#define TERM_Y_COUNT TERM_X_COUNT

const int l_subcount[L_COUNT] = {64,34,20,7,3,1};
const int b_subcount[B_COUNT] = {5,2};
const int r_subcount[R_COUNT] = {40,10,6,2,1};

/////////////////////////////////////////////////
///  Earth Periodic Terms
/////////////////////////////////////////////////
const double L_TERMS[L_COUNT][L_MAX_SUBCOUNT][TERM_COUNT]=
{
    {
        {175347046.0,0,0},
        {3341656.0,4.6692568,6283.07585},
        {34894.0,4.6261,12566.1517},
        {3497.0,2.7441,5753.3849},
        {3418.0,2.8289,3.5231},
        {3136.0,3.6277,77713.7715},
        {2676.0,4.4181,7860.4194},
        {2343.0,6.1352,3930.2097},
        {1324.0,0.7425,11506.7698},
        {1273.0,2.0371,529.691},
        {1199.0,1.1096,1577.3435},
        {990,5.233,5884.927},
        {902,2.045,26.298},
        {857,3.508,398.149},
        {780,1.179,5223.694},
        {753,2.533,5507.553},
        {505,4.583,18849.228},
        {492,4.205,775.523},
        {357,2.92,0.067},
        {317,5.849,11790.629},
        {284,1.899,796.298},
        {271,0.315,10977.079},
        {243,0.345,5486.778},
        {206,4.806,2544.314},
        {205,1.869,5573.143},
        {202,2.458,6069.777},
        {156,0.833,213.299},
        {132,3.411,2942.463},
        {126,1.083,20.775},
        {115,0.645,0.98},
        {103,0.636,4694.003},
        {102,0.976,15720.839},
        {102,4.267,7.114},
```

```
    {99,6.21,2146.17},
    {98,0.68,155.42},
    {86,5.98,161000.69},
    {85,1.3,6275.96},
    {85,3.67,71430.7},
    {80,1.81,17260.15},
    {79,3.04,12036.46},
    {75,1.76,5088.63},
    {74,3.5,3154.69},
    {74,4.68,801.82},
    {70,0.83,9437.76},
    {62,3.98,8827.39},
    {61,1.82,7084.9},
    {57,2.78,6286.6},
    {56,4.39,14143.5},
    {56,3.47,6279.55},
    {52,0.19,12139.55},
    {52,1.33,1748.02},
    {51,0.28,5856.48},
    {49,0.49,1194.45},
    {41,5.37,8429.24},
    {41,2.4,19651.05},
    {39,6.17,10447.39},
    {37,6.04,10213.29},
    {37,2.57,1059.38},
    {36,1.71,2352.87},
    {36,1.78,6812.77},
    {33,0.59,17789.85},
    {30,0.44,83996.85},
    {30,2.74,1349.87},
    {25,3.16,4690.48}
},
{
    {628331966747.0,0,0},
    {206059.0,2.678235,6283.07585},
    {4303.0,2.6351,12566.1517},
    {425.0,1.59,3.523},
    {119.0,5.796,26.298},
    {109.0,2.966,1577.344},
    {93,2.59,18849.23},
    {72,1.14,529.69},
    {68,1.87,398.15},
    {67,4.41,5507.55},
    {59,2.89,5223.69},
    {56,2.17,155.42},
    {45,0.4,796.3},
    {36,0.47,775.52},
    {29,2.65,7.11},
    {21,5.34,0.98},
    {19,1.85,5486.78},
    {19,4.97,213.3},
    {17,2.99,6275.96},
    {16,0.03,2544.31},
    {16,1.43,2146.17},
    {15,1.21,10977.08},
    {12,2.83,1748.02},
    {12,3.26,5088.63},
    {12,5.27,1194.45},
    {12,2.08,4694},
    {11,0.77,553.57},
    {10,1.3,6286.6},
    {10,4.24,1349.87},
    {9,2.7,242.73},
    {9,5.64,951.72},
    {8,5.3,2352.87},
    {6,2.65,9437.76},
    {6,4.67,4690.48}
},
{
    {52919.0,0,0},
    {8720.0,1.0721,6283.0758},
    {309.0,0.867,12566.152},
    {27,0.05,3.52},
    {16,5.19,26.3},
    {16,3.68,155.42},
    {10,0.76,18849.23},
```

```
        {9,2.06,77713.77},
        {7,0.83,775.52},
        {5,4.66,1577.34},
        {4,1.03,7.11},
        {4,3.44,5573.14},
        {3,5.14,796.3},
        {3,6.05,5507.55},
        {3,1.19,242.73},
        {3,6.12,529.69},
        {3,0.31,398.15},
        {3,2.28,553.57},
        {2,4.38,5223.69},
        {2,3.75,0.98}
    },
    {
        {289.0,5.844,6283.076},
        {35,0,0},
        {17,5.49,12566.15},
        {3,5.2,155.42},
        {1,4.72,3.52},
        {1,5.3,18849.23},
        {1,5.97,242.73}
    },
    {
        {114.0,3.142,0},
        {8,4.13,6283.08},
        {1,3.84,12566.15}
    },
    {
        {1,3.14,0}
    }
};

const double B_TERMS[B_COUNT][B_MAX_SUBCOUNT][TERM_COUNT]=
{
    {
        {280.0,3.199,84334.662},
        {102.0,5.422,5507.553},
        {80,3.88,5223.69},
        {44,3.7,2352.87},
        {32,4,1577.34}
    },
    {
        {9,3.9,5507.55},
        {6,1.73,5223.69}
    }
};

const double R_TERMS[R_COUNT][R_MAX_SUBCOUNT][TERM_COUNT]=
{
    {
        {100013989.0,0,0},
        {1670700.0,3.0984635,6283.07585},
        {13956.0,3.05525,12566.1517},
        {3084.0,5.1985,77713.7715},
        {1628.0,1.1739,5753.3849},
        {1576.0,2.8469,7860.4194},
        {925.0,5.453,11506.77},
        {542.0,4.564,3930.21},
        {472.0,3.661,5884.927},
        {346.0,0.964,5507.553},
        {329.0,5.9,5223.694},
        {307.0,0.299,5573.143},
        {243.0,4.273,11790.629},
        {212.0,5.847,1577.344},
        {186.0,5.022,10977.079},
        {175.0,3.012,18849.228},
        {110.0,5.055,5486.778},
        {98,0.89,6069.78},
        {86,5.69,15720.84},
        {86,1.27,161000.69},
        {65,0.27,17260.15},
        {63,0.92,529.69},
        {57,2.01,83996.85},
        {56,5.24,71430.7},
        {49,3.25,2544.31},
```

```
        {47,2.58,775.52},
        {45,5.54,9437.76},
        {43,6.01,6275.96},
        {39,5.36,4694},
        {38,2.39,8827.39},
        {37,0.83,19651.05},
        {37,4.9,12139.55},
        {36,1.67,12036.46},
        {35,1.84,2942.46},
        {33,0.24,7084.9},
        {32,0.18,5088.63},
        {32,1.78,398.15},
        {28,1.21,6286.6},
        {28,1.9,6279.55},
        {26,4.59,10447.39}
    },
    {
        {103019.0,1.10749,6283.07585},
        {1721.0,1.0644,12566.1517},
        {702.0,3.142,0},
        {32,1.02,18849.23},
        {31,2.84,5507.55},
        {25,1.32,5223.69},
        {18,1.42,1577.34},
        {10,5.91,10977.08},
        {9,1.42,6275.96},
        {9,0.27,5486.78}
    },
    {
        {4359.0,5.7846,6283.0758},
        {124.0,5.579,12566.152},
        {12,3.14,0},
        {9,3.63,77713.77},
        {6,1.87,5573.14},
        {3,5.47,18849.23}
    },
    {
        {145.0,4.273,6283.076},
        {7,3.92,12566.15}
    },
    {
        {4,2.56,6283.08}
    }
};

//////////////////////////////////////////////////////////////////
///  Periodic Terms for the nutation in longitude and obliquity
//////////////////////////////////////////////////////////////////

const int Y_TERMS[Y_COUNT][TERM_Y_COUNT]=
{
    {0,0,0,0,1},
    {-2,0,0,2,2},
    {0,0,0,2,2},
    {0,0,0,0,2},
    {0,1,0,0,0},
    {0,0,1,0,0},
    {-2,1,0,2,2},
    {0,0,0,2,1},
    {0,0,1,2,2},
    {-2,-1,0,2,2},
    {-2,0,1,0,0},
    {-2,0,0,2,1},
    {0,0,-1,2,2},
    {2,0,0,0,0},
    {0,0,1,0,1},
    {2,0,-1,2,2},
    {0,0,-1,0,1},
    {0,0,1,2,1},
    {-2,0,2,0,0},
    {0,0,-2,2,1},
    {2,0,0,2,2},
    {0,0,2,2,2},
    {0,0,2,0,0},
    {-2,0,1,2,2},
    {0,0,0,2,0},
```

```
    {-2,0,0,2,0},
    {0,0,-1,2,1},
    {0,2,0,0,0},
    {2,0,-1,0,1},
    {-2,2,0,2,2},
    {0,1,0,0,1},
    {-2,0,1,0,1},
    {0,-1,0,0,1},
    {0,0,2,-2,0},
    {2,0,-1,2,1},
    {2,0,1,2,2},
    {0,1,0,2,2},
    {-2,1,1,0,0},
    {0,-1,0,2,2},
    {2,0,0,2,1},
    {2,0,1,0,0},
    {-2,0,2,2,2},
    {-2,0,1,2,1},
    {2,0,-2,0,1},
    {2,0,0,0,1},
    {0,-1,1,0,0},
    {-2,-1,0,2,1},
    {-2,0,0,0,1},
    {0,0,2,2,1},
    {-2,0,2,0,1},
    {-2,1,0,2,1},
    {0,0,1,-2,0},
    {-1,0,1,0,0},
    {-2,1,0,0,0},
    {1,0,0,0,0},
    {0,0,1,2,0},
    {0,0,-2,2,2},
    {-1,-1,1,0,0},
    {0,1,1,0,0},
    {0,-1,1,2,2},
    {2,-1,-1,2,2},
    {0,0,3,2,2},
    {2,-1,0,2,2},
};

const double PE_TERMS[Y_COUNT][TERM_PE_COUNT]={
    {-171996,-174.2,92025,8.9},
    {-13187,-1.6,5736,-3.1},
    {-2274,-0.2,977,-0.5},
    {2062,0.2,-895,0.5},
    {1426,-3.4,54,-0.1},
    {712,0.1,-7,0},
    {-517,1.2,224,-0.6},
    {-386,-0.4,200,0},
    {-301,0,129,-0.1},
    {217,-0.5,-95,0.3},
    {-158,0,0,0},
    {129,0.1,-70,0},
    {123,0,-53,0},
    {63,0,0,0},
    {63,0.1,-33,0},
    {-59,0,26,0},
    {-58,-0.1,32,0},
    {-51,0,27,0},
    {48,0,0,0},
    {46,0,-24,0},
    {-38,0,16,0},
    {-31,0,13,0},
    {29,0,0,0},
    {29,0,-12,0},
    {26,0,0,0},
    {-22,0,0,0},
    {21,0,-10,0},
    {17,-0.1,0,0},
    {16,0,-8,0},
    {-16,0.1,7,0},
    {-15,0,9,0},
    {-13,0,7,0},
    {-12,0,6,0},
    {11,0,0,0},
    {-10,0,5,0},
```

```
    {-8,0,3,0},
    {7,0,-3,0},
    {-7,0,0,0},
    {-7,0,3,0},
    {-7,0,3,0},
    {6,0,0,0},
    {6,0,-3,0},
    {6,0,-3,0},
    {-6,0,3,0},
    {-6,0,3,0},
    {5,0,0,0},
    {-5,0,3,0},
    {-5,0,3,0},
    {-5,0,3,0},
    {4,0,0,0},
    {4,0,0,0},
    {4,0,0,0},
    {-4,0,0,0},
    {-4,0,0,0},
    {-4,0,0,0},
    {3,0,0,0},
    {-3,0,0,0},
    {-3,0,0,0},
    {-3,0,0,0},
    {-3,0,0,0},
    {-3,0,0,0},
    {-3,0,0,0},
    {-3,0,0,0},
};

////////////////////////////////////////////////

double rad2deg(double radians)
{
    return (180.0/PI)*radians;
}

double deg2rad(double degrees)
{
    return (PI/180.0)*degrees;
}

double limit_degrees(double degrees)
{
    double limited;

    degrees /= 360.0;
    limited = 360.0*(degrees-floor(degrees));
    if (limited < 0) limited += 360.0;

    return limited;
}

double limit_degrees180pm(double degrees)
{
    double limited;

    degrees /= 360.0;
    limited = 360.0*(degrees-floor(degrees));
    if      (limited < -180.0) limited += 360.0;
    else if (limited >  180.0) limited -= 360.0;

    return limited;
}

double limit_degrees180(double degrees)
{
    double limited;

    degrees /= 180.0;
    limited = 180.0*(degrees-floor(degrees));
    if (limited < 0) limited += 180.0;

    return limited;
}
```

```
double limit_zero2one(double value)
{
    double limited;

    limited = value - floor(value);
    if (limited < 0) limited += 1.0;

    return limited;
}

double limit_minutes(double minutes)
{
    double limited=minutes;

    if      (limited < -20.0) limited += 1440.0;
    else if (limited >  20.0) limited -= 1440.0;

    return limited;
}

double dayfrac_to_local_hr(double dayfrac, double timezone)
{
    return 24.0*limit_zero2one(dayfrac + timezone/24.0);
}

double third_order_polynomial(double a, double b, double c, double d, double x)
{
    return ((a*x + b)*x + c)*x + d;
}

//////////////////////////////////////////////////////////////////////////////////////////
int validate_inputs(spa_data *spa)
{
    if ((spa->year        < -2000) || (spa->year        > 6000)) return 1;
    if ((spa->month       < 1    ) || (spa->month       > 12  )) return 2;
    if ((spa->day         < 1    ) || (spa->day         > 31  )) return 3;
    if ((spa->hour        < 0    ) || (spa->hour        > 24  )) return 4;
    if ((spa->minute      < 0    ) || (spa->minute      > 59  )) return 5;
    if ((spa->second      < 0    ) || (spa->second      > 59  )) return 6;
    if ((spa->pressure    < 0    ) || (spa->pressure    > 5000)) return 12;
    if ((spa->temperature <= -273) || (spa->temperature > 6000)) return 13;
    if ((spa->hour        == 24  ) && (spa->minute      > 0   )) return 5;
    if ((spa->hour        == 24  ) && (spa->second      > 0   )) return 6;

    if (fabs(spa->delta_t)       > 8000    ) return 7;
    if (fabs(spa->timezone)      > 12      ) return 8;
    if (fabs(spa->longitude)     > 180     ) return 9;
    if (fabs(spa->latitude)      > 90      ) return 10;
    if (fabs(spa->atmos_refract) > 5       ) return 16;
    if (     spa->elevation      < -6500000) return 11;

    if ((spa->function == SPA_ZA_INC) || (spa->function == SPA_ALL))
    {
        if (fabs(spa->slope)         > 360) return 14;
        if (fabs(spa->azm_rotation)  > 360) return 15;
    }

    return 0;
}
//////////////////////////////////////////////////////////////////////////////////////////
double julian_day (int year, int month, int day, int hour, int minute, int second, double tz)
{
    double day_decimal, julian_day, a;

    day_decimal = day + (hour - tz + (minute + second/60.0)/60.0)/24.0;

    if (month < 3) {
        month += 12;
        year--;
    }

    julian_day = floor(365.25*(year+4716.0)) + floor(30.6001*(month+1)) + day_decimal - 1524.5;

    if (julian_day > 2299160.0) {
        a = floor(year/100);
        julian_day += (2 - a + floor(a/4));
```

```
    }

    return julian_day;
}

double julian_century(double jd)
{
    return (jd-2451545.0)/36525.0;
}

double julian_ephemeris_day(double jd, double delta_t)
{
    return jd+delta_t/86400.0;
}

double julian_ephemeris_century(double jde)
{
    return (jde - 2451545.0)/36525.0;
}

double julian_ephemeris_millennium(double jce)
{
    return (jce/10.0);
}

double earth_periodic_term_summation(const double terms[][TERM_COUNT], int count, double jme)
{
    int i;
    double sum=0;

    for (i = 0; i < count; i++)
        sum += terms[i][TERM_A]*cos(terms[i][TERM_B]+terms[i][TERM_C]*jme);

    return sum;
}

double earth_values(double term_sum[], int count, double jme)
{
    int i;
    double sum=0;

    for (i = 0; i < count; i++)
        sum += term_sum[i]*pow(jme, i);

    sum /= 1.0e8;

    return sum;
}

double earth_heliocentric_longitude(double jme)
{
    double sum[L_COUNT];
    int i;

    for (i = 0; i < L_COUNT; i++)
        sum[i] = earth_periodic_term_summation(L_TERMS[i], l_subcount[i], jme);

    return limit_degrees(rad2deg(earth_values(sum, L_COUNT, jme)));

}

double earth_heliocentric_latitude(double jme)
{
    double sum[B_COUNT];
    int i;

    for (i = 0; i < B_COUNT; i++)
        sum[i] = earth_periodic_term_summation(B_TERMS[i], b_subcount[i], jme);

    return rad2deg(earth_values(sum, B_COUNT, jme));

}

double earth_radius_vector(double jme)
{
    double sum[R_COUNT];
```

```
    int i;

    for (i = 0; i < R_COUNT; i++)
        sum[i] = earth_periodic_term_summation(R_TERMS[i], r_subcount[i], jme);

    return earth_values(sum, R_COUNT, jme);

}

double geocentric_longitude(double l)
{
    double theta = l + 180.0;

    if (theta >= 360.0) theta -= 360.0;

    return theta;
}

double geocentric_latitude(double b)
{
    return -b;
}

double mean_elongation_moon_sun(double jce)
{
    return third_order_polynomial(1.0/189474.0, -0.0019142, 445267.11148, 297.85036, jce);
}

double mean_anomaly_sun(double jce)
{
    return third_order_polynomial(-1.0/300000.0, -0.0001603, 35999.05034, 357.52772, jce);
}

double mean_anomaly_moon(double jce)
{
    return third_order_polynomial(1.0/56250.0, 0.0086972, 477198.867398, 134.96298, jce);
}

double argument_latitude_moon(double jce)
{
    return third_order_polynomial(1.0/327270.0, -0.0036825, 483202.017538, 93.27191, jce);
}

double ascending_longitude_moon(double jce)
{
    return third_order_polynomial(1.0/450000.0, 0.0020708, -1934.136261, 125.04452, jce);
}

double xy_term_summation(int i, double x[TERM_X_COUNT])
{
    int j;
    double sum=0;

    for (j = 0; j < TERM_Y_COUNT; j++)
        sum += x[j]*Y_TERMS[i][j];

    return sum;
}

void nutation_longitude_and_obliquity(double jce, double x[TERM_X_COUNT], double *del_psi,
                                                                           double *del_epsilon)
{
    int i;
    double xy_term_sum, sum_psi=0, sum_epsilon=0;

    for (i = 0; i < Y_COUNT; i++) {
        xy_term_sum  = deg2rad(xy_term_summation(i, x));
        sum_psi     += (PE_TERMS[i][TERM_PSI_A] + jce*PE_TERMS[i][TERM_PSI_B])*sin(xy_term_sum);
        sum_epsilon += (PE_TERMS[i][TERM_EPS_C] + jce*PE_TERMS[i][TERM_EPS_D])*cos(xy_term_sum);
    }

    *del_psi     = sum_psi     / 36000000.0;
    *del_epsilon = sum_epsilon / 36000000.0;
}

double ecliptic_mean_obliquity(double jme)
```

```
{
    double u = jme/10.0;

    return 84381.448 + u*(-4680.96 + u*(-1.55 + u*(1999.25 + u*(-51.38 + u*(-249.67 +
                      u*(  -39.05 + u*( 7.12 + u*(  27.87 + u*(  5.79 + u*2.45)))))))));
}

double ecliptic_true_obliquity(double delta_epsilon, double epsilon0)
{
    return delta_epsilon + epsilon0/3600.0;
}

double aberration_correction(double r)
{
    return -20.4898 / (3600.0*r);
}

double apparent_sun_longitude(double theta, double delta_psi, double delta_tau)
{
    return theta + delta_psi + delta_tau;
}

double greenwich_mean_sidereal_time (double jd, double jc)
{
    return limit_degrees(280.46061837 + 360.98564736629 * (jd - 2451545.0) +
                                        jc*jc*(0.000387933 - jc/38710000.0));
}

double greenwich_sidereal_time (double nu0, double delta_psi, double epsilon)
{
    return nu0 + delta_psi*cos(deg2rad(epsilon));
}

double geocentric_sun_right_ascension(double lamda, double epsilon, double beta)
{
    double lamda_rad   = deg2rad(lamda);
    double epsilon_rad = deg2rad(epsilon);

    return limit_degrees(rad2deg(atan2(sin(lamda_rad)*cos(epsilon_rad) -
                                        tan(deg2rad(beta))*sin(epsilon_rad), cos(lamda_rad))));
}

double geocentric_sun_declination(double beta, double epsilon, double lamda)
{
    double beta_rad    = deg2rad(beta);
    double epsilon_rad = deg2rad(epsilon);

    return rad2deg(asin(sin(beta_rad)*cos(epsilon_rad) +
                        cos(beta_rad)*sin(epsilon_rad)*sin(deg2rad(lamda))));
}

double observer_hour_angle(double nu, double longitude, double alpha_deg)
{
    return limit_degrees(nu + longitude - alpha_deg);
}

double sun_equatorial_horizontal_parallax(double r)
{
    return 8.794 / (3600.0 * r);
}

void sun_right_ascension_parallax_and_topocentric_dec(double latitude, double elevation,
               double xi, double h, double delta, double *delta_alpha, double *delta_prime)
{
    double delta_alpha_rad;
    double lat_rad   = deg2rad(latitude);
    double xi_rad    = deg2rad(xi);
    double h_rad     = deg2rad(h);
    double delta_rad = deg2rad(delta);
    double u = atan(0.99664719 * tan(lat_rad));
    double y = 0.99664719 * sin(u) + elevation*sin(lat_rad)/6378140.0;
    double x =              cos(u) + elevation*cos(lat_rad)/6378140.0;

    delta_alpha_rad =      atan2(               - x*sin(xi_rad) *sin(h_rad),
                                 cos(delta_rad) - x*sin(xi_rad) *cos(h_rad));
```

```
    *delta_prime = rad2deg(atan2((sin(delta_rad) - y*sin(xi_rad))*cos(delta_alpha_rad),
                                 cos(delta_rad) - x*sin(xi_rad) *cos(h_rad)));

    *delta_alpha = rad2deg(delta_alpha_rad);
}

double topocentric_sun_right_ascension(double alpha_deg, double delta_alpha)
{
    return alpha_deg + delta_alpha;
}

double topocentric_local_hour_angle(double h, double delta_alpha)
{
    return h - delta_alpha;
}

double topocentric_elevation_angle(double latitude, double delta_prime, double h_prime)
{
    double lat_rad         = deg2rad(latitude);
    double delta_prime_rad = deg2rad(delta_prime);

    return rad2deg(asin(sin(lat_rad)*sin(delta_prime_rad) +
                        cos(lat_rad)*cos(delta_prime_rad) * cos(deg2rad(h_prime))));
}

double atmospheric_refraction_correction(double pressure, double temperature,
                                         double atmos_refract, double e0)
{
    double del_e = 0;

    if (e0 >= -1*(SUN_RADIUS + atmos_refract))
        del_e = (pressure / 1010.0) * (283.0 / (273.0 + temperature)) *
                1.02 / (60.0 * tan(deg2rad(e0 + 10.3/(e0 + 5.11))));

    return del_e;
}

double topocentric_elevation_angle_corrected(double e0, double delta_e)
{
    return e0 + delta_e;
}

double topocentric_zenith_angle(double e)
{
    return 90.0 - e;
}

double topocentric_azimuth_angle_neg180_180(double h_prime, double latitude, double delta_prime)
{
    double h_prime_rad = deg2rad(h_prime);
    double lat_rad     = deg2rad(latitude);

    return rad2deg(atan2(sin(h_prime_rad),
                         cos(h_prime_rad)*sin(lat_rad) -
                         tan(deg2rad(delta_prime))*cos(lat_rad)));
}

double topocentric_azimuth_angle_zero_360(double azimuth180)
{
    return azimuth180 + 180.0;
}

double surface_incidence_angle(double zenith, double azimuth180, double azm_rotation,
                                                                  double slope)
{
    double zenith_rad = deg2rad(zenith);
    double slope_rad  = deg2rad(slope);

    return rad2deg(acos(cos(zenith_rad)*cos(slope_rad)  +
                        sin(slope_rad )*sin(zenith_rad) * cos(deg2rad(azimuth180 -
                        azm_rotation))));
}

double sun_mean_longitude(double jme)
{
    return limit_degrees(280.4664567 + jme*(360007.6982779 + jme*(0.03032028 +
```

```
                    jme*(1/49931.0    + jme*(-1/15300.0     + jme*(-1/2000000.0))))));
}

double eot(double m, double alpha, double del_psi, double epsilon)
{
    return limit_minutes(4.0*(m - 0.0057183 - alpha + del_psi*cos(deg2rad(epsilon))));
}

double approx_sun_transit_time(double alpha_zero, double longitude, double nu)
{
    return (alpha_zero - longitude - nu) / 360.0;
}

double sun_hour_angle_at_rise_set(double latitude, double delta_zero, double h0_prime)
{
    double h0             = -99999;
    double latitude_rad   = deg2rad(latitude);
    double delta_zero_rad = deg2rad(delta_zero);
    double argument       = (sin(deg2rad(h0_prime)) - sin(latitude_rad)*sin(delta_zero_rad)) /
                                                      (cos(latitude_rad)*cos(delta_zero_rad));

    if (fabs(argument) <= 1) h0 = limit_degrees180(rad2deg(acos(argument)));

    return h0;
}

void approx_sun_rise_and_set(double *m_rts, double h0)
{
    double h0_dfrac = h0/360.0;

    m_rts[SUN_RISE]    = limit_zero2one(m_rts[SUN_TRANSIT] - h0_dfrac);
    m_rts[SUN_SET]     = limit_zero2one(m_rts[SUN_TRANSIT] + h0_dfrac);
    m_rts[SUN_TRANSIT] = limit_zero2one(m_rts[SUN_TRANSIT]);
}

double rts_alpha_delta_prime(double *ad, double n)
{
    double a = ad[JD_ZERO] - ad[JD_MINUS];
    double b = ad[JD_PLUS] - ad[JD_ZERO];

    if (fabs(a) >= 2.0) a = limit_zero2one(a);
    if (fabs(b) >= 2.0) b = limit_zero2one(b);

    return ad[JD_ZERO] + n * (a + b + (b-a)*n)/2.0;
}

double rts_sun_altitude(double latitude, double delta_prime, double h_prime)
{
    double latitude_rad    = deg2rad(latitude);
    double delta_prime_rad = deg2rad(delta_prime);

    return rad2deg(asin(sin(latitude_rad)*sin(delta_prime_rad) +
                        cos(latitude_rad)*cos(delta_prime_rad)*cos(deg2rad(h_prime))));
}

double sun_rise_and_set(double *m_rts,   double *h_rts,   double *delta_prime, double latitude,
                        double *h_prime, double h0_prime, int sun)
{
    return m_rts[sun] + (h_rts[sun] - h0_prime) /

          (360.0*cos(deg2rad(delta_prime[sun]))*cos(deg2rad(latitude))
          *sin(deg2rad(h_prime[sun])));
}

//////////////////////////////////////////////////////////////////////////////////////////
// Calculate required SPA parameters to get the right ascension (alpha) and declination (delta)
// Note: JD must be already calculated and in structure
//////////////////////////////////////////////////////////////////////////////////////////
void calculate_geocentric_sun_right_ascension_and_declination(spa_data *spa)
{
    double x[TERM_X_COUNT];

    spa->jc = julian_century(spa->jd);

    spa->jde = julian_ephemeris_day(spa->jd, spa->delta_t);
    spa->jce = julian_ephemeris_century(spa->jde);
```

```
    spa->jme = julian_ephemeris_millennium(spa->jce);

    spa->l = earth_heliocentric_longitude(spa->jme);
    spa->b = earth_heliocentric_latitude(spa->jme);
    spa->r = earth_radius_vector(spa->jme);

    spa->theta = geocentric_longitude(spa->l);
    spa->beta  = geocentric_latitude(spa->b);

    x[TERM_X0] = spa->x0 = mean_elongation_moon_sun(spa->jce);
    x[TERM_X1] = spa->x1 = mean_anomaly_sun(spa->jce);
    x[TERM_X2] = spa->x2 = mean_anomaly_moon(spa->jce);
    x[TERM_X3] = spa->x3 = argument_latitude_moon(spa->jce);
    x[TERM_X4] = spa->x4 = ascending_longitude_moon(spa->jce);

    nutation_longitude_and_obliquity(spa->jce, x, &(spa->del_psi), &(spa->del_epsilon));

    spa->epsilon0 = ecliptic_mean_obliquity(spa->jme);
    spa->epsilon  = ecliptic_true_obliquity(spa->del_epsilon, spa->epsilon0);

    spa->del_tau   = aberration_correction(spa->r);
    spa->lamda     = apparent_sun_longitude(spa->theta, spa->del_psi, spa->del_tau);
    spa->nu0       = greenwich_mean_sidereal_time (spa->jd, spa->jc);
    spa->nu        = greenwich_sidereal_time (spa->nu0, spa->del_psi, spa->epsilon);

    spa->alpha = geocentric_sun_right_ascension(spa->lamda, spa->epsilon, spa->beta);
    spa->delta = geocentric_sun_declination(spa->beta, spa->epsilon, spa->lamda);
}

//////////////////////////////////////////////////////////////////////////
// Calculate Equation of Time (EOT) and Sun Rise, Transit, & Set (RTS)
//////////////////////////////////////////////////////////////////////////

void calculate_eot_and_sun_rise_transit_set(spa_data *spa)
{
    spa_data sun_rts = *spa;
    double nu, m, h0, n;
    double alpha[JD_COUNT], delta[JD_COUNT];
    double m_rts[SUN_COUNT], nu_rts[SUN_COUNT], h_rts[SUN_COUNT];
    double alpha_prime[SUN_COUNT], delta_prime[SUN_COUNT], h_prime[SUN_COUNT];
    double h0_prime = -1*(SUN_RADIUS + spa->atmos_refract);
    int i;


    m        = sun_mean_longitude(spa->jme);
    spa->eot = eot(m, spa->alpha, spa->del_psi, spa->epsilon);

    sun_rts.hour = sun_rts.minute = sun_rts.second = sun_rts.timezone = 0;

    sun_rts.jd = julian_day (sun_rts.year, sun_rts.month,  sun_rts.day,
                             sun_rts.hour, sun_rts.minute, sun_rts.second, sun_rts.timezone);

    calculate_geocentric_sun_right_ascension_and_declination(&sun_rts);
    nu = sun_rts.nu;

    sun_rts.delta_t = 0;
    sun_rts.jd--;
    for (i = 0; i < JD_COUNT; i++) {
        calculate_geocentric_sun_right_ascension_and_declination(&sun_rts);
        alpha[i] = sun_rts.alpha;
        delta[i] = sun_rts.delta;
        sun_rts.jd++;
    }

    m_rts[SUN_TRANSIT] = approx_sun_transit_time(alpha[JD_ZERO], spa->longitude, nu);
    h0 = sun_hour_angle_at_rise_set(spa->latitude, delta[JD_ZERO], h0_prime);

    if (h0 >= 0) {

        approx_sun_rise_and_set(m_rts, h0);

        for (i = 0; i < SUN_COUNT; i++) {

            nu_rts[i]      = nu + 360.985647*m_rts[i];

            n              = m_rts[i] + spa->delta_t/86400.0;
```

```
            alpha_prime[i] = rts_alpha_delta_prime(alpha, n);
            delta_prime[i] = rts_alpha_delta_prime(delta, n);

            h_prime[i]     = limit_degrees180pm(nu_rts[i] + spa->longitude - alpha_prime[i]);

            h_rts[i]       = rts_sun_altitude(spa->latitude, delta_prime[i], h_prime[i]);
        }

        spa->srha = h_prime[SUN_RISE];
        spa->ssha = h_prime[SUN_SET];
        spa->sta  = h_rts[SUN_TRANSIT];

        spa->suntransit = dayfrac_to_local_hr(m_rts[SUN_TRANSIT] - h_prime[SUN_TRANSIT] / 360.0,
                                              spa->timezone);

        spa->sunrise = dayfrac_to_local_hr(sun_rise_and_set(m_rts, h_rts, delta_prime,
                        spa->latitude, h_prime, h0_prime, SUN_RISE), spa->timezone);

        spa->sunset  = dayfrac_to_local_hr(sun_rise_and_set(m_rts, h_rts, delta_prime,
                        spa->latitude, h_prime, h0_prime, SUN_SET),  spa->timezone);

    } else spa->srha= spa->ssha= spa->sta= spa->suntransit= spa->sunrise= spa->sunset= -99999;

}

//////////////////////////////////////////////////////////////////////////////////////////
// Calculate all SPA parameters and put into structure
// Note: All inputs values (listed in header file) must already be in structure
//////////////////////////////////////////////////////////////////////////////////////////
int spa_calculate(spa_data *spa)
{
    int result;

    result = validate_inputs(spa);

    if (result == 0)
    {
        spa->jd = julian_day (spa->year, spa->month,  spa->day,
                              spa->hour, spa->minute, spa->second, spa->timezone);

        calculate_geocentric_sun_right_ascension_and_declination(spa);

        spa->h  = observer_hour_angle(spa->nu, spa->longitude, spa->alpha);
        spa->xi = sun_equatorial_horizontal_parallax(spa->r);

        sun_right_ascension_parallax_and_topocentric_dec(spa->latitude, spa->elevation, spa->xi,
                                  spa->h, spa->delta, &(spa->del_alpha), &(spa->delta_prime));

        spa->alpha_prime = topocentric_sun_right_ascension(spa->alpha, spa->del_alpha);
        spa->h_prime     = topocentric_local_hour_angle(spa->h, spa->del_alpha);

        spa->e0      = topocentric_elevation_angle(spa->latitude, spa->delta_prime,
        spa->h_prime);
        spa->del_e   = atmospheric_refraction_correction(spa->pressure, spa->temperature,
                                                         spa->atmos_refract, spa->e0);
        spa->e       = topocentric_elevation_angle_corrected(spa->e0, spa->del_e);

        spa->zenith     = topocentric_zenith_angle(spa->e);
        spa->azimuth180 = topocentric_azimuth_angle_neg180_180(spa->h_prime, spa->latitude,
                                                               spa->delta_prime);
        spa->azimuth    = topocentric_azimuth_angle_zero_360(spa->azimuth180);

        if ((spa->function == SPA_ZA_INC) || (spa->function == SPA_ALL))
            spa->incidence  = surface_incidence_angle(spa->zenith, spa->azimuth180,
                                                      spa->azm_rotation, spa->slope);

        if ((spa->function == SPA_ZA_RTS) || (spa->function == SPA_ALL))
            calculate_eot_and_sun_rise_transit_set(spa);
    }

    return result;
}
//////////////////////////////////////////////////////////////////////////////////////////
```