Contents

Programming interface to the Swiss Ephemeris

This document describes the proprietary programmer's interface to the Swiss Ephemeris library.

The Swiss Ephemeris is made available by its authors under a dual licensing system. The software developer, who uses any part of Swiss Ephemeris in his or her software, must choose between one of the two license models, which are:

To use Swiss Ephemeris, the licensing conditions imposed by Astrodienst for Swiss Ephemeris must be fulfilled. A copy of the license file is found in file LICENSE.

Please note: Since Swiss Ephemeris release 2.10.01 the GPL license has been replaced with the AGPL license, as one of the options in Astrodienst's dual licensing model.

[toc]

1. The programming steps to get a planet's position

To compute a celestial body or point with SWISSEPH, you have to do the following steps (use swetest.c as an example). The details of the functions will be explained in the following chapters.

1. Set the directory path of the ephemeris files, e.g.:

```
swe_set_ephe_path("C:\SWEPH\EPHE");
```

2. From the birth date, compute the Julian day number:

```
jul_day_UT = swe_julday(year, month, day, hour, gregflag);
```

3. Compute a planet or other bodies:

```
ret_flag = swe_calc_ut(jul_day_UT, planet_no, flag, lon_lat_rad, err_msg);
```

or a fixed star:

```
ret_flag = swe_fixstar_ut(star_nam, jul_day_UT, flag, lon_lat_rad, err_msg);
```

NOTE:

The functions **swe_calc_ut()** and **swe_fixstar_ut()** were introduced with Swisseph version 1.60.

If you use a Swisseph version older than 1.60 or if you want to work with Ephemeris Time, you have to proceed as follows instead:

- first, if necessary, convert universal time (UT) to ephemeris time (ET):

```
jul_day_ET = jul_day_UT + swe_deltat(jul_day_UT);
```

- then compute a planet or other bodies:

```
ret_flag = swe_calc(jul_day_ET, planet_no, flag, lon_lat_rad, err_msg);
```

- or a fixed star:

```
ret_flag = swe_fixstar(star_nam, jul_day_ET, flag, lon_lat_rad, err_msg);
```

4. At the end of your computations close all files and free memory calling **swe_close();**

Here is a miniature sample program, it is in the source distribution as swemini.c:

```
#include "swephexp.h" /* this includes "sweodef.h" */
int main()
{
  char *sp, sdate[AS_MAXCH], snam[40], serr[AS_MAXCH];
  int jday = 1, jmon = 1, jyear = 2000;
  double jut = 0.0;
  double tjd_ut, te, x2[6];
  long iflag, iflgret;
  int p;
```

```
  swe_set_ephe_path(NULL);
  iflag = SEFLG_SPEED;
  while (TRUE) {
    printf("nDate (d.m.y) ?");
    gets(sdate);
    // stop if a period . is entered
    if (*sdate == '.') return OK;
    // we have day, month and year and convert to Julian day number
    tjd_ut = swe_julday(jyear, jmon, jday, jut, SE_GREG_CAL);
    printf("date: %02d.%02d.%d at 0:00 Universal timen", jday, jmon, jyear);
    printf("planet tlongitudetlatitudetdistancetspeed long.n");
    // a loop over all planets
    for (p = SE_SUN; p <= SE_CHIRON; p++) {
      if (p == SE_EARTH) continue;
      // do the coordinate calculation for this planet p
      iflgret = swe_calc_ut(tjd_ut, p, iflag, x2, serr);
      /* if there is a problem, a negative value is returned and an
      * error message is in serr.
      */
      if (iflgret < 0) {
    printf("error: %sn", serr);
    return ERR;
      }
      // get the name of the planet p
      swe_get_planet_name(p, snam);
      // print the coordinates
      printf("%10st%11.7ft%10.7ft%10.7ft%10.7fn", snam, x2[0], x2[1], x2[2], x2[3]);
    }
  }
  return OK;
}
```

2. The Ephemeris file related functions

2.1. swe_set_ephe_path()

This is the first function that should be called before any other function of the Swiss Ephemeris. Even if you don't want to set an ephemeris path and use the Moshier ephemeris, it is nevertheless recommended to call **swe_set_ephe_path**(NULL), because this function makes important initializations. If you don't do that, the Swiss Ephemeris may work, but the results may be not 100% consistent.

If the environment variable SE_EPHE_PATH exists in the environment where Swiss Ephemeris is used, its content is used to find the ephemeris files. The variable can contain a directory name, or a list of directory names separated by ; (semicolon) on Windows or : (colon) on Unix.

```
void swe_set_ephe_path(char *path);
```

Usually an application will want to set its own ephemeris, e.g. as follows:

```
swe_set_ephe_path("C:\SWEPH\EPHE");
```

The argument can be a single directory name or a list of directories, which are then searched in sequence. The argument of this call is ignored if the environment variable SE_EPHE_PATH exists and is not empty. If you want to make sure that your program overrides any environment variable setting, you can use **putenv()** to set it to an empty string.

If the path is longer than 256 bytes, `swe_set_ephe_path()` sets the path \SWEPH\EPHE instead.

If no environment variable exists and `swe_set_ephe_path()` is never called, the built-in ephemeris path is used. On Windows it is "\sweph\ephe" relative to the current working drive, on Unix it is "/home/ephe".

Asteroid ephemerides are looked for in the subdirectories ast0, ast1, ast2 .. ast9 of the ephemeris directory and, if not found there, in the ephemeris directory itself. Asteroids with numbers 0 – 999 are expected in directory ast0, those with numbers 1000 – 1999 in directory ast1 etc.

The environment variable SE_EPHE_PATH is most convenient when a user has several applications installed which all use the Swiss Ephemeris but would normally expect the ephemeris files in different application-specific directories. The use can override this by setting the environment variable, which forces all the different applications to use the same ephemeris directory. This allows him to use only one set of installed ephemeris files for all different applications. A developer should accept this override feature and allow the sophisticated users to exploit it.

2.2. swe_close()

```
void swe_close( void);
```

At the end of your computations you can release all resources (open files and allocated memory) used by the Swiss Ephemeris DLL.

After **swe_close()**, **no** Swiss Ephemeris functions should be used unless you call **swe_set_ephe_path()** again and, if required, **swe_set_jpl_file()**.

2.3. swe_set_jpl_file()

```
/* set name of JPL ephemeris file */

void swe_set_jpl_file(char *fname);
```

If you work with the JPL ephemeris, SwissEph uses the default file name which is defined in swephexp.h as SE_FNAME_DFT. Currently, it has the "de431.eph".

If a different JPL ephemeris file is required, call the function **swe_set_jpl_file()** to make the file name known to the software, e.g.

swe_set_jpl_file("de430.eph");

This file must reside in the ephemeris path you are using for all your ephemeris files.

If the file name is longer than 256 byte, **swe_set_jpl_file()** cuts the file name to a length of 256 bytes. The error will become visible after the first call of **swe_calc()**, when it will return zero positions and an error message.

2.4. swe_version()

```
/* find out version number of your Swiss Ephemeris version */

char *swe_version(char *svers);
/* svers is a string variable with sufficient space to contain the
version number (255 char) */
```

The function returns a pointer to the string svers, i.e. to the version number of the Swiss Ephemeris that your software is using.

2.5. swe_get_library_path()

```
/* find out the library path of the DLL or executable */

char *swe_get_library_path(char *spath);

/* spath is a string variable with sufficient space to contain the
library path (255 char) */
```

The function returns a pointer to the string spath, which contains the path in which the executable resides. If it is running with a DLL, then spath contains the path of the DLL.

2.6. swe_get_current_file_data()

This is function can be used to find out the start and end date of an *se1 ephemeris file after a call of swe_calc().

The function returns data from internal file structures sweph.fidat used in the *last call* to swe_calc() or swe_fixstar(). Data returned are (currently) 0 with JPL files and fixed star files. Thus, the function is only useful for ephemerides of planets or asteroids that are based on *.se1 files.

```
const char *swe_get_current_file_data(int ifno, double *tfstart, double *tfend, int *denum);
// ifno = 0 planet file sepl_xxx, used for Sun .. Pluto, or jpl file
// ifno = 1 moon file semo_xxx
// ifno = 2 main asteroid file seas_xxx if such an object was computed
// ifno = 3 other asteroid or planetary moon file, if such object was computed
// ifno = 4 star file
// Return value: full file pathname, or NULL if no data
// tfstart = start date of file,
// tfend = end data of fila,
// denum = jpl ephemeris number 406 or 431 from which file was derived
// all three return values are zero for a jpl file or a star file.
```

3. Planetary Positions:

Before calling one of the functions below or any other Swiss Ephemeris function, **it is strongly recommended** to call the function **swe_set_ephe_path()**. Even if you don't want to set an ephemeris path and use the Moshier ephemeris, it is nevertheless recommended to call **swe_set_ephe_path**(NULL), because **this function makes important initializations**. If you don't do that, the Swiss Ephemeris may work but the results may be not 100% consistent.

3.1. The functions swe_calc_ut(), swe_calc(), and swe_calc_pctr()

swe_calc_ut() computes positions of planets, in Universal Time UT, asteroids, lunar nodes and apogees. It works geocentric or heliocentric, or in some other modes controlled by parameter iflag.

```
int32 swe_calc_ut(
  double tjd_ut, // Julian day number, Universal Time
  int32 ipl,     // planet number
  int32 iflag,   // flag bits
  double *xx,    // return array for 6 position values
  char *serr     // 256 bytes for optional error string
);
```

More details on the parameters

- tjd_ut = Julian day, representing date and time in Universal Time
- ipl = body number, as defined in section **3.2.**
- iflag = a 32 bit integer containing bit flags that indicate what kind of computation is wanted, see section **3.3.**
- xx = array of 6 doubles for longitude, latitude, distance, speed in long., speed in lat., and speed in dist.
- serr[256] = character string to return error messages in case of error. If it is NULL, no error details are returned.

Return value: see section **3.5.**

swe_calc() computes the same, with time expressed in Ephemeris Time (ET), now commonly called Terrestial Time (TT).

```
int32 swe_calc(
  double tjd_et, // Julian day number, Ephemeris Time
  int32 ipl,     // planet number
  int32 iflag,   // flag bits
  double *xx,    // return array for 6 position values
  char *serr     // 256 bytes for optional error string
);
```

The relationship between UT and ET is: tjd_et = tjd_ut + swe_deltat(tjd_ut)

======= ### The call parameters

swe_calc_ut() was introduced with Swisseph **version 1.60** and makes planetary calculations a bit simpler. For the steps required, see the chapter The programming steps to get a planet's position.

swe_calc_ut() and **swe_calc()** work exactly the same way except that **swe_calc()** requires Ephemeris Time (more accurate: Terrestrial Time (TT)) as a parameter whereas **swe_calc_ut()** expects Universal Time (UT). For common astrological calculations, you will only need **swe_calc_ut()** and will not have to think any more about the conversion between Universal Time and Ephemeris Time.

swe_calc_ut() and **swe_calc()** compute positions of planets, asteroids, lunar nodes and apogees. They are defined as follows:

```
int32 **swe_calc_ut**(double tjd_ut, int32 ipl, int32 iflag, double *xx, char *serr);

// tjd_ut = [Julian day], Universal Time
// ipl = body number
// iflag = a 32 bit integer containing bit flags that indicate what kind of computation is wanted
// xx = array of 6 doubles for longitude, latitude, distance, speed in long., speed in lat., and speed
// serr[256] = character string to return error messages in case of error.
```

and

```
int32 swe_calc_ut(double tjd_et, int32 ipl, int32 iflag, double *xx, char *serr);
```

same but

tjd_et = Julian day, Ephemeris time, where tjd_et = tjd_ut + swe_deltat(tjd_ut)

A detailed description of these variables will be given in the following sections.

swe_calc_pctr() calculates planetocentric positions of planets, i. e. positions as observed from some different planet, e.g. Jupiter-centric ephemerides. The function can actually calculate any object as observed from any other object, e.g. also the position of some asteroid as observed from another asteroid or from a planetary moon. The function declaration is as follows:

```
int32 swe_calc_pctr(
  double tjd_et, // input julian day number in TT
  int32 ipl,    // target object
  int32 iplctr, // center object
  int32 iflag,  // flag bits, as with swe_calc()
  double *xx,   // return array for 6 position values
  char *serr    // 256 bytes for optional error string
);
```

3.2. Bodies (int ipl)

To tell **swe_calc_ut()** or **swe_calc()** which celestial body or factor should be computed, a fixed set of body numbers is used. The body numbers are defined in swephexp.h:

```
/* planet numbers for the ipl parameter in swe_calc() */
#define SE_ECL_NUT -1
#define SE_SUN 0
#define SE_MOON 1
#define SE_MERCURY 2
#define SE_VENUS 3
#define SE_MARS 4
#define SE_JUPITER 5
```

```
#define SE_SATURN 6
#define SE_URANUS 7
#define SE_NEPTUNE 8
#define SE_PLUTO 9
#define SE_MEAN_NODE 10
#define SE_TRUE_NODE 11
#define SE_MEAN_APOG 12
#define SE_OSCU_APOG 13
#define SE_EARTH 14
#define SE_CHIRON 15
#define SE_PHOLUS 16
#define SE_CERES 17
#define SE_PALLAS 18
#define SE_JUNO 19
#define SE_VESTA 20
#define SE_INTP_APOG 21
#define SE_INTP_PERG 22
#define SE_NPLANETS 23
#define SE_FICT_OFFSET 40 // offset for fictitious objects
#define SE_NFICT_ELEM 15
#define SE_PLMOON_OFFSET 9000 // offset for planetary moons
#define SE_AST_OFFSET 10000 // offset for asteroids
/* Hamburger or Uranian "planets" */
#define SE_CUPIDO 40
#define SE_HADES 41
#define SE_ZEUS 42
#define SE_KRONOS 43
#define SE_APOLLON 44
#define SE_ADMETOS 45
#define SE_VULKANUS 46
#define SE_POSEIDON 47
/* other fictitious bodies */
#define SE_ISIS 48
#define SE_NIBIRU 49
#define SE_HARRINGTON 50
#define SE_NEPTUNE_LEVERRIER 51
#define SE_NEPTUNE_ADAMS 52
#define SE_PLUTO_LOWELL 53
#define SE_PLUTO_PICKERING 54
```

Additional asteroids

Body numbers of other asteroids are above SE_AST_OFFSET (= 10000) and have to be constructed as follows:

```
ipl = SE_AST_OFFSET + minor_planet_catalogue_number;
```

e.g. Eros:

```
ipl = SE_AST_OFFSET + 433;  // (= 10433)
```

The names of the asteroids and their catalogue numbers can be found in seasnam.txt.

Examples are:

```
5 Astraea
6 Hebe
```

```
7 Iris
8 Flora
9 Metis
10 Hygiea
30 Urania
42 Isis not identical with "Isis-Transpluto"
153 Hilda has an own asteroid belt at 4 AU
227 Philosophia
251 Sophia
259 Aletheia
275 Sapientia
279 Thule asteroid close to Jupiter
375 Ursula
433 Eros
763 Cupido different from Witte's Cupido
944 Hidalgo
1181 Lilith not identical with Dark Moon 'Lilith'
1221 Amor
1387 Kama
1388 Aphrodite
1862 Apollo different from Witte's Apollon
3553 Damocles highly eccentric orbit between Mars and Uranus
3753 Cruithne "second moon" of Earth
4341 Poseidon Greek Neptune - different from Witte's Poseidon
4464 Vulcano fire god - different from Witte's Vulkanus and
7066 Nessus third named Centaur - between Saturn and Pluto
```

There are two ephemeris files for each asteroid (except the main asteroids), a long one and a short one:

- se09999.se1 long-term ephemeris of asteroid number 9999, 3000 BCE – 3000 CE
- se09999s.se1 short ephemeris of asteroid number 9999, 1500 – 2100 CE

The larger file is about 10 times the size of the short ephemeris. If the user does not want an ephemeris for the time before 1500 he might prefer to work with the short files. If so, just copy the files ending with "s.se1" to your hard disk. **swe_calc()** tries the long one and on failure automatically takes the short one.

Asteroid ephemerides are looked for in the subdirectories ast0, ast1, ast2 .. ast9 etc. of the ephemeris directory and, if not found there, in the ephemeris directory itself. Asteroids with numbers 0 – 999 are expected in directory ast0, those with numbers 1000 – 1999 in directory ast1 etc.

Note that not all asteroids can be computed for the whole period of Swiss Ephemeris. The orbits of some of them are extremely sensitive to perturbations by major planets. E.g. **CHIRON**, cannot be computed for the time before **650 CE** and after **4650 CE** because of close encounters with Saturn. Outside this time range, Swiss Ephemeris returns the error code, an error message, and a position value 0. Be aware, that the user will **have to handle** this case in his program. Computing Chiron transits for Jesus or Alexander the Great will not work.

The same is true for Pholus before **3850 BCE**, and for many other asteroids, as e.g. 1862 Apollo. He becomes chaotic before the year **1870 CE**, when he approaches Venus very closely. Swiss Ephemeris does not provide positions of Apollo for earlier centuries !

NOTE on asteroid names:

Asteroid names are listed in the file seasnam.txt. This file is in the ephemeris directory.

Planetary moons and body centers

Ephemerides of planetary moons and centers of body (COB) were introduced with Swiss Ephemeris version 2.10.

Their Swiss Ephemeris body numbers are between SE_PLMOON_OFFSET (= 9000) and SE_AST_OFFSET (= 10000) and are constructed as follows:

```
ipl = SE_PLMOON_OFFSET + planet_number * 100 + moon_number_in_JPL_Horizons;
```

e.g., Jupiter moon Io:

```
ipl = SE_PLMOON_OFFSET + SE_JUPITER * 100 + 1;  // (= 9501)
```

Centers of body (COB) are calculated the same way, i.e. like a planetary moon but with the "moon number" 99;

e.g. Jupiter center of body:

```
ipl = SE_PLMOON_OFFSET + SE_JUPITER * 100 + 99; //(= 9599)
```

- Moons of Mars: 9401 – 9402
- Moons of Jupiter: 9501 – 95xx; Center of body: 9599
- Moons of Saturn: 9601 – 96xx; Center of body: 9699
- Moons of Uranus: 9701 – 97xx; Center of body: 9799
- Moons of Neptune: 9801 – 98xx; Center of body: 9899
- Moons of Pluto: 9901 – 99xx; Center of body: 9999

A full list of existing planetary moons is found here:

https://en.wikipedia.org/wiki/List_of_natural_satellites .

The ephemeris files of the planetary moons and COB are in **the subdirectory sat.** Like the subdirectories of asteroids, the directory sat must be created in the path which is defined using the function swe_set_ephe_path().

The ephemeris files can be downloaded from here:

https://www.astro.com/ftp/swisseph/ephe/sat/.

The list of objects available in the Swiss Ephemeris is:

```
9401 Phobos/Mars
9402 Deimos/Mars
9501 Io/Jupiter
9502 Europa/Jupiter
9503 Ganymede/Jupiter
9504 Callisto/Jupiter
9599 Jupiter/COB
9601 Mimas/Saturn
9602 Enceladus/Saturn
9603 Tethys/Saturn
9604 Dione/Saturn
9605 Rhea/Saturn
9606 Titan/Saturn
9607 Hyperion/Saturn
9608 Iapetus/Saturn
9699 Saturn/COB
9701 Ariel/Uranus
9702 Umbriel/Uranus
9703 Titania/Uranus
9704 Oberon/Uranus
9705 Miranda/Uranus
```

```
9799 Uranus/COB
9801 Triton/Neptune
9802 Triton/Nereid
9808 Proteus/Neptune
9899 Neptune/COB
9901 Charon/Pluto
9902 Nix/Pluto
9903 Hydra/Pluto
9904 Kerberos/Pluto
9905 Styx/Pluto
9999 Pluto/COB
```

The maximum differences between barycenter and center of body (COB) are:

```
Mars (0.2 m, irrelevant to us)
Jupiter 0.075 arcsec (jd 2468233.5)
Saturn 0.053 arcsec (jd 2463601.5)
Uranus 0.0032 arcsec (jd 2446650.5)
Neptune 0.0036 arcsec (jd 2449131.5)
Pluto 0.088 arcsec (jd 2437372.5)
```

(from one-day-step calculations over 150 years)

If you prefer using COB rather than barycenters, you should understand that:

- The performance is not as good for COB as for barycenters. With transit calculations you could run into troubles.
- The ephemerides are limited to the time range 1900 to 2047.

Fictitious planets

Fictitious planets have numbers greater than or equal to 40. The user can define his or her own fictitious planets. The orbital elements of these planets must be written into the file seorbel.txt. The function **swe_calc()** looks for the file seorbel.txt in the ephemeris path set by **swe_set_ephe_path()**. If no orbital elements file is found, **swe_calc()** uses the built-in orbital elements of the above mentioned [Uranian planets] and some other bodies. The planet number of a fictitious planet is defined as

```
ipl = SE_FICT_OFFSET_1 + number_of_elements_set;
```

e.g. for Kronos: ipl = 39 + 4 = 43.

The file seorbel.txt contains the orbital elements for fictitious bodies. It is described in Appendix A.

Obliquity and nutation

A special body number SE_ECL_NUT is provided to compute the obliquity of the ecliptic and the nutation. Of course nutation is already added internally to the planetary coordinates by **swe_calc()** but sometimes it will be needed as a separate value.

iflgret = **swe_calc**(tjd_et, SE_ECL_NUT, 0, x, serr);

x is an array of 6 doubles as usual. They will be filled as follows:

x[0] = true obliquity of the Ecliptic (includes nutation)

x[1] = mean obliquity of the Ecliptic

x[2] = nutation in longitude

x[3] = nutation in obliquity

x[4] = x[5] = 0

3.3. Options chosen by flag bits (int32 iflag)

The use of flag bits

If no bits are set, i.e. if iflag == 0, **swe_calc()** computes what common astrological ephemerides (as available in book shops) supply, i.e. an [apparent] body position in **geocentric** ecliptic polar coordinates (longitude, latitude, and distance) relative to the true equinox of the date.

If the speed of the body is required, set iflag = SEFLG_SPEED.

For mathematical points as the mean lunar node and the mean apogee, there is no apparent position. **swe_calc()** returns true positions for these points.

If you need another kind of computation, use the flags explained in the following paragraphs (c.f. swephexp.h). Their names begin with ‚SEFLG_'. To combine them, you have to concatenate them (inclusive-or) as in the following example:

```
iflag = SEFLG_SPEED | SEFLG_TRUEPOS; // (or: iflag = SEFLG_SPEED +
SEFLG_TRUEPOS;) // C

iflag = SEFLG_SPEED or SEFLG_TRUEPOS; // (or: iflag = SEFLG_SPEED +
SEFLG_TRUEPOS;) // Pascal
```

With this value of iflag, **swe_calc()** will compute true positions (i.e. not accounted for light-time) with speed.

The flag bits, which are defined in swephexp.h, are:

```
#define SEFLG_JPLEPH 1L // use JPL ephemeris
#define SEFLG_SWIEPH 2L // use SWISSEPH ephemeris, default
#define SEFLG_MOSEPH 4L // use Moshier ephemeris
#define SEFLG_HELCTR 8L // return heliocentric position
#define SEFLG_TRUEPOS 16L // return true positions, not apparent
#define SEFLG_J2000 32L // no precession, i.e. give J2000 equinox
#define SEFLG_NONUT 64L // no nutation, i.e. mean equinox of date
#define SEFLG_SPEED3 128L // speed from 3 positions (**do not use it**, SEFLG_SPEED is faster and more p
#define SEFLG_SPEED 256L // high precision speed (analyt. comp.)
#define SEFLG_NOGDEFL 512L // turn off gravitational deflection
#define SEFLG_NOABERR 1024L // turn off 'annual' aberration of light
#define SEFLG_ASTROMETRIC (SEFLG_NOABERR|SEFLG_NOGDEFL) // astrometric positions
#define SEFLG_EQUATORIAL 2048L // equatorial positions are wanted
#define SEFLG_XYZ 4096L // cartesian, not polar, coordinates
#define SEFLG_RADIANS 8192L // coordinates in radians, not degrees
#define SEFLG_BARYCTR 16384L // barycentric positions
#define SEFLG_TOPOCTR (32*1024L) // topocentric positions
#define SEFLG_SIDEREAL (64*1024L) // sidereal positions
#define SEFLG_ICRS (128*1024L) // ICRS (DE406 reference frame)
#define SEFLG_DPSIDEPS_1980 (256*1024) /* reproduce JPL Horizons * 1962 - today to 0.002 arcsec. */
#define SEFLG_JPLHOR SEFLG_DPSIDEPS_1980
#define SEFLG_JPLHOR_APPROX (512*1024) /* approximate JPL Horizons 1962 - today */
#define SEFLG_CENTER_BODY (1024*1024) /* calculate position of center of body (COB) of planet, not baryc
```

Note, COB can be calculated either

- ipl = SE_JUPITER with iflag |= SEFLG_CENTER_BODY or

- ipl = 9599 (= 9000 + SE_JUPITER * 100 + 99) without any additional bit in iflag

Ephemeris flags

The flags to choose an ephemeris are: (s. swephexp.h)

```
SEFLG_JPLEPH /* use JPL ephemeris */
SEFLG_SWIEPH /* use Swiss Ephemeris */
SEFLG_MOSEPH /* use Moshier ephemeris */
```

If none of this flags is specified, **swe_calc()** tries to compute the default ephemeris. The default ephemeris is defined in swephexp.h:

```
#define SEFLG_DEFAULTEPH SEFLG_SWIEPH
```

In this case the default ephemeris is Swiss Ephemeris. If you have not specified an ephemeris in iflag, **swe_calc()** tries to compute a Swiss Ephemeris position. If it does not find the required Swiss Ephemeris file either, it computes a Moshier position.

Speed flag

Swe_calc() does not compute speed if you do not add the speed flag SEFLG_SPEED. E.g.

```
iflag |= SEFLG_SPEED;
```

The computation of speed is usually cheap, so you may set this bit by default even if you do not need the speed.

Coordinate systems, degrees and radians

- SEFLG_EQUATORIAL returns equatorial positions: right ascension and declination.
- SEFLG_XYZ returns x, y, z coordinates instead of longitude, latitude, and distance.
- SEFLG_RADIANS returns position in radians, not degrees.

E.g. to compute right ascension and declination, write:

```
iflag = SEFLG_SWIEPH | SEFLG_SPEED | SEFLG_EQUATORIAL;
```

NOTE concerning equatorial coordinates: With sidereal modes SE_SIDM_J2000, SE_SIDM_B1950, SE_SIDM_J1900, SE_SIDM_GALALIGN_MARDYKS or if the sidereal flag SE_SIDBIT_ECL_T0 is set, the function provides right ascension and declination relative to the mean equinox of the reference epoch (J2000, B1950, J1900, etc.).

With other sidereal modes or ayanamshas right ascension and declination are given relative to the mean equinox of date.

Specialties (going beyond common interest)

True or apparent positions

Common ephemerides supply apparent geocentric positions. Since the journey of the light from a planet to the Earth takes some time, the planets are never seen where they actually are, but where they were a few minutes or hours before. Astrology uses to work with the positions **we see**. (More precisely: with the positions we would see, if we stood at the center of the Earth and could see the sky. Actually, the geographical position of the observer could be of importance as well and topocentric positions could be computed, but

this is usually not taken into account in astrology.). The geocentric position for the Earth (SE_EARTH) is returned as zero.

To compute the **true** geometrical position of a planet, disregarding light-time, you have to add the flag SEFLG_TRUEPOS.

Topocentric positions

To compute topocentric positions, i.e. positions referred to the place of the observer (the birth place) rather than to the center of the Earth, do as follows:

- call **swe_set_topo**(geo_lon, geo_lat, altitude_above_sea) (The geographic longitude and latitude must be in degrees, the altitude in meters.)
- add the flag SEFLG_TOPOCTR to iflag
- call **swe_calc**(...)

Heliocentric positions

To compute a heliocentric position, add SEFLG_HELCTR.

A heliocentric position can be computed for all planets including the moon. For the sun, lunar nodes and lunar apogees the coordinates are returned as zero; **no error message appears**.

Barycentric positions

SEFLG_BARYCTR yields coordinates as referred to the solar system barycenter. However, this option **is not completely implemented**. It was used for program tests during development. It works only with the JPL and the Swiss Ephemeris, **not with the Moshier** ephemeris; and **only with physical bodies**, but not with the nodes and the apogees.

Moreover, the barycentric Sun of Swiss Ephemeris has "only" a precision of 0.1". Higher accuracy would have taken a lot of storage, on the other hand it is not needed for precise geocentric and heliocentric positions. For more precise barycentric positions the JPL ephemeris file should be used.

A barycentric position can be computed for all planets including the sun and moon. For the lunar nodes and lunar apogees the coordinates are returned as zero; no error message appears.

Astrometric positions

For astrometric positions, which are sometimes given in the Astronomical Almanac, the light-time correction is computed, but annual aberration and the light-deflection by the sun neglected. This can be done with SEFLG_NOABERR and SEFLG_NOGDEFL. For positions related to the mean equinox of 2000, you must set SEFLG_J2000 and SEFLG_NONUT, as well.

True or mean equinox of date

swe_calc() usually computes the positions as referred to the true equinox of the date (i.e. with nutation). If you want the mean equinox, you can turn nutation off, using the flag bit SEFLG_NONUT.

J2000 positions and positions referred to other equinoxes

swe_calc() usually computes the positions as referred to the equinox of date. SEFLG_J2000 yields data referred to the equinox J2000. For positions referred to other equinoxes, SEFLG_SIDEREAL has to be set and the equinox specified by **swe_set_sid_mode()**. For more information, read the description of this function.

Sidereal positions

To compute sidereal positions, set bit SEFLG_SIDEREAL and use the function **swe_set_sid_mode()** in order to define the ayanamsha you want. For more information, read the description of this function.

JPL Horizons positions

For apparent positions of the planets, JPL Horizons follows a different approach from Astronomical Almanac and from the IERS Conventions 2003 and 2010. It uses the old precession models IAU 1976 (Lieske) and nutation IAU 1980 (Wahr) and corrects the resulting positions by adding daily-measured celestial pole offsets (delta_psi and delta_epsilon) to nutation. (IERS Conventions 1996, p. 22) While this approach is more accurate in some respect, it is not referred to the same reference frame. For more details see the general documentation of the Swiss Ephemeris in http://www.astro.com/swisseph/swisseph.htm, ch. 2.1.2.2.

Apparent positions of JPL Horizons can be reproduced with about 0.001 arcsec precision using the flag SEFLG_JPLHOR. For best accuracy, the daily Earth orientation parameters (EOP) delta_psi and delta_eps relative to the IAU 1980 precession/nutation model must be downloaded and saved in the ephemeris path defined by swe_set_ephe_path(). The EOP files are found on the IERS website:

http://www.iers.org/IERS/EN/DataProducts/EarthOrientationData/eop.html

The following files are required:

- EOP 08 C04 (IAU1980) - one file (1962-now) http://datacenter.iers.org/eop/-/somos/5Rgv/document/tx14iers.0z9/eopc04_08.62-now Put this file into your ephemeris path and rename it as "eop_1962_today.txt".
- finals.data (IAU1980) http://datacenter.iers.org/eop/-/somos/5Rgv/document/tx14iers.0q0/finals.data Put this file into your ephemeris path, too, and rename it as "eop_finals.txt".

If the Swiss Ephemeris does not find these files, it defaults to SEFLG_JPLHORA, which is a very good approximation of Horizons, at least for 1962 to present.

SEFLG_JPLHORA can be used independently for the whole time range of the Swiss Ephemeris.

Note, the Horizons mode works only with planets and fixed stars. With lunar nodes and apsides, we use our standard methods.

3.4. Position and Speed (double xx[6])

swe_calc() returns the coordinates of position and velocity in the following order:

index	**Ecliptic position**	**Equatorial position (SEFLG_EQUATORIAL)**
0	Longitude	right ascension
1	Latitude	declination
2	Distance in AU	distance in AU
3	Speed in longitude (deg/day)	speed in right ascension (deg/day)
4	Speed in latitude (deg/day)	speed in declination (deg/day)
5	Speed in distance (AU/day)	speed in distance (AU/day)

If you need rectangular coordinates (SEFLG_XYZ), **swe_calc()** returns x, y, z, dx, dy, dz in AU.

Once you have computed a planet, e.g., in ecliptic coordinates, its equatorial position or its rectangular coordinates are available, too. You can get them very cheaply (little CPU time used), calling again **swe_calc()** with the same parameters, but adding SEFLG_EQUATORIAL or SEFLG_XYZ to iflag, **swe_calc()** will not compute the body again, just return the data specified from internal storage.

3.5. Error handling and return values

swe_calc() (as well as **swe_calc_ut()**, **swe_fixstar()**, and **swe_fixstar_ut()**) returns a 32-bit integer value. This value is >= 0, if the function call was successful, and < 0, if a fatal error has occurred. In addition an error string or a warning can be returned in the string parameter serr.

A **fatal error code (< 0)** and an error string are returned in one of the following cases:

- if an illegal [body number] has been specified;
- if a Julian day beyond the ephemeris limits has been specified;
- if the length of the ephemeris file is not correct (damaged file);
- on read error, e.g. a file index points to a position beyond file length (data on file are corrupt);
- if the copyright section in the ephemeris file has been destroyed.

If any of these errors occurs:

- the return code of the function is -1;
- the position and speed variables are set to zero;
- the type of error is indicated in the error string serr.

On success, the return code contains flag bits that indicate what kind of computation has been done. This value will usually be equal to iflag, however sometimes may differ from it. If an option specified by iflag cannot be fulfilled or makes no sense, **swe_calc** just does what can be done. E.g., if you specify that you want JPL ephemeris, but **swe_calc** cannot find the ephemeris file, it tries to do the computation with any available ephemeris. The ephemeris actually used will be indicated in the return value of **swe_calc**. So, to make sure that **swe_calc()** has found the ephemeris required, you may want to check, e.g.:

if (return_code > 0 && (return_code & SEFLG_JPLEPH))

However, usually it should be sufficient to do the ephemeris test once only, at the very beginning of the program.

In such cases, there is also a warning in the error string serr, saying that:

warning: SwissEph file 'sepl_18.se1' not found in PATH '...' ; using Moshier eph.;

Apart from that, positive values of return_code need not be checked, but maybe useful for debugging purposes or for understanding what exactly has been done by the function.

Some flags may be removed, if they are incompatible with other flags, e.g.:

- if two or more ephemerides (SEFLG_JPLEPH, SEFLG_SWIEPH, SEFLG_MOSEPH) are combined.
- if the topocentric flag (SEFLG_TOPOCTR) is combined with the heliocentric (SEFLG_HELCTR) or the barycentric flag (SEFLG_BARYCTR).
- etc.

Some flags may be added in the following cases:

- If no ephemeris flag was specified, the return value contains SEFLG_SWIEPH;
- With J2000 calculations (SEFLG_J2000) or other sidereal calculations (SEFLG_SIDEREAL), the no-nutation flag (SEFLG_NONUT) is added;
- With heliocentric (SEFLG_HELCTR) and barycentric (SEFLG_BARYCTR) calculations, the flags for "no aberration" (SEFLG_NOABERR) and "no light deflection" (SEFLG_NOGDEFL) are added.

4. Other functions for planet and asteroid data

4.1 swe_get_planet_name()

This function allows to find a planetary or asteroid name, when the planet number is given. The function definition is:

```
char* swe_get_planet_name( int32 ipl, char *spname);
```

If an asteroid name is wanted, the function does the following:

- The name is first looked for in the asteroid file.
- Because many asteroids, especially the ones with high catalogue numbers, have no names yet (or have only a preliminary designation like 1968 HB), and because the Minor Planet Center of the IAU add new names quite often, it happens that there is no name in the asteroid file although the asteroid has already been given a name. For this, we have the file seasnam.txt, a file that contains a list of all named asteroid and is usually more up to date. If **swe_calc()** finds a preliminary designation, it looks for a name in this file.

4.2. swe_get_orbital_elements() (Kepler elements and orbital data)

This function calculates osculating elements (Kepler elements) and orbital periods for a planet, the Earth-Moon barycenter, or an asteroid. The elements are calculated relative to the mean ecliptic J2000.

The elements define the orbital ellipse under the premise that it is a two-body system and there are no perturbations from other celestial bodies. The elements are particularly bad for the Moon, which is strongly perturbed by the Sun. It is not recommended to calculate ephemerides using Kepler elements.

Important: This function should not be used for ephemerides of the perihelion or aphelion of a planet. Note that when the position of a perihelion is calculated using swe_get_orbital_elements(), this position is **not** measured on the ecliptic, but on the orbit of the planet itself, thus it is **not** an ecliptic position. Also note that the positions of the nodes are always calculated relative to the mean equinox 2000 and never precessed to the ecliptic or equator of date. For ecliptic positions of a perihelion or aphelion or a node, you should use the function swe_nod_aps() or swe_nod_aps_ut().

```
int32 swe_get_orbital_elements(
  double tjd_et, // input date in TT (Julian day number)
  int32 ipl,     // planet number
  int32 iflag,   // flag bits, see detailed docu
  double *dret,  // return values, see detailed docu below
  char *serr
```

```
);

/* Function calculates osculating orbital elements (Kepler elements) of a planet
  or asteroid or the EMB. The function returns error,
  if called for the Sun, the lunar nodes, or the apsides.
  Input parameters:
  tjd_et Julian day number, in TT (ET)
  ipl object number
  iflag can contain
  - ephemeris flag: SEFLG_JPLEPH, SEFLG_SWIEPH, SEFLG_MOSEPH
  - center:
  Sun: SEFLG_HELCTR (assumed as default) or
  SS Barycentre: SEFLG_BARYCTR (rel. to solar system barycentre)
  (only possible for planets beyond Jupiter)
  For elements of the Moon, the calculation is geocentric.
  - sum all masses inside the orbit to be computed (method
  of Astronomical Almanac):
  SEFLG_ORBEL_AA
  - reference ecliptic: SEFLG_J2000;
  if missing, mean ecliptic of date is chosen (still not implemented)
  output parameters:
  dret[] array of return values, declare as dret[50]
  dret[0] semimajor axis (a)
  dret[1] eccentricity (e)
  dret[2] inclination (in)
  dret[3] longitude of ascending node (upper case omega OM)
  dret[4] argument of periapsis (lower case omega om)
  dret[5] longitude of periapsis (peri)
  dret[6] mean anomaly at epoch (M0)
  dret[7] true anomaly at epoch (N0)
  dret[8] eccentric anomaly at epoch (E0)
  dret[9] mean longitude at epoch (LM)
  dret[10] sidereal orbital period in tropical years
  dret[11] mean daily motion
  dret[12] tropical period in years
  dret[13] synodic period in days,
  negative, if inner planet (Venus, Mercury, Aten asteroids) or Moon
  dret[14] time of perihelion passage
  dret[15] perihelion distance
  dret[16] aphelion distance
*/
```

4.3. swe_orbit_max_min_true_distance()

This function calculates the maximum possible distance, the minimum possible distance, and the current true distance of planet, the EMB, or an asteroid. The calculation can be done either heliocentrically or geocentrically. With heliocentric calculations, it is based on the momentary Kepler ellipse of the planet. With geocentric calculations, it is based on the Kepler ellipses of the planet and the EMB. The geocentric calculation is rather expensive..

```
int32 swe_orbit_max_min_true_distance(
  double tjd_et, // input date in TT (Julian day number)
  int32 ipl,     // planet number
```

```
  int32 iflag,   // flag bits, see detailed docu
  double *dmax,  // return value: maximum distance based on osculating elements
  double *dmin,  // return value: minimum distance based on osculating elements
  double *dtrue, // return value: current distance
  char *serr
);

/* Input:
  tjd_et epoch
  ipl planet number
  iflag ephemeris flag and optional heliocentric flag (SEFLG_HELCTR)

  output:
  dmax maximum distance (pointer to double)
  dmin minimum distance (pointer to double)
  dtrue true distance (pointer to double)
  serr error string
*/
```

4.4. Planetary Apsides and Nodes, swe_nod_aps_ut() and swe_nod_aps()

The functions **swe_nod_aps_ut()** and **swe_nod_aps()** compute planetary nodes and apsides (perihelia, aphelia, second focal points of the orbital ellipses). Both functions do exactly the same except that they expect a different time parameter (cf. **swe_calc_ut()** and **swe_calc()**).

The definitions are:

```
  int32 swe_nod_aps_ut(
    double tjd_ut, // Julian day number in UT
    int32 ipl,     // planet number
    int32 iflag,   // flag bits
    int32 method,  // method, see explanations below
    double *xnasc, // array of 6 double for ascending node
    double *xndsc, // array of 6 double for descending node
    double *xperi, // array of 6 double for perihelion
    double *xaphe, // array of 6 double for aphelion
    char *serr     // character string to contain error messages, 256 chars
  );

  int32 **swe_nod_aps**(
    double tjd_et, // Julian day number in TT
    int32 ipl,
    int32 iflag,
    int32 method,
    double *xnasc,
    double *xndsc,
    double *xperi,
    double *xaphe,
    char *serr
  );
```

The parameter iflag allows the same specifications as with the function **swe_calc_ut()**. I.e., it contains the Ephemeris flag, the heliocentric, topocentric, speed, nutation flags etc. etc.

The parameter method tells the function what kind of nodes or apsides are required:

```
#define SE_NODBIT_MEAN 1
```

Mean nodes and apsides are calculated for the bodies that have them, i.e. for the Moon and the planets Mercury through Neptune, osculating ones for Pluto and the asteroids. This is the default method, also used if method=0.

```
#define SE_NODBIT_OSCU 2
```

Osculating nodes and apsides are calculated for all bodies.

```
#define SE_NODBIT_OSCU_BAR 4
```

Osculating nodes and apsides are calculated for all bodies. With planets beyond Jupiter, the nodes and apsides are calculated from *barycentric* positions and speed. Cf. the explanations in swisseph.doc.

If this bit is combined with SE_NODBIT_MEAN, mean values are given for the planets Mercury - Neptune.

```
#define SE_NODBIT_FOPOINT 256
```

The second focal point of the orbital ellipse is computed and returned in the array of the aphelion. This bit can be combined with any other bit.

4.5 swe_solcross() and swe_solcross_ut()

These functions find the crossing of the Sun over a given ecliptic position:

```
double swe_solcross_ut(
  double x2cross,   // longitude to cross
  double tjd_ut,    // start time for search
  int32 iflag,      // flag bits
  char *serr
);

double swe_solcross(
  double x2cross,   // longitude to cross
  double tjd_et,    // start time for search
  int32 iflag,      // flag bits
  char *serr
);
```

Return value: double jx = time of next crossing, in Ephemeris Time or Universal Time.

In case of error, a value of jx < tjd is returned. Because the crossing search is always forward in time, returning an earlier time is an indication of error. In addition, string serr will contain error details.

The precision is 1 milliarcsecond, i.e. at the returned time the Sun is closer than 0.001 arcsec to x2cross.

These flag bits in iflag can be useful:

```
SEFLG_TRUEPOS

SEFLG_NONUT

SEFLG_EQUATORIAL (x2cross is a rectascension value, a point on the equator, and not on the ecliptic)
```

4.6. swe_mooncross() and swe_mooncross_ut()

These functions find the crossing of the Moon over a given ecliptic position:

```
double swe_mooncross_ut(
  double x2cross,   // longitude to cross
  double tjd_ut,    // start time for search
  int32 iflag,      // flag bits
  char *serr
);

double swe_mooncross(
  double x2cross,   // longitude to cross
  double tjd_et,    // start time for search
  int32 iflag,      // flag bits
  char *serr
);
```

Return value: double jx = time of next crossing, in Ephemeris Time or Universal Time.

In case of error, a value of jx < tjd is returned. Because the crossing search is always forward in time, returning an earlier time is an indication of error. In addition, string serr will contain error details (unless it is a NULL pointer)

The precision is 1 milliarcsecond, i.e. at the returned time the Moon is closer than 0.001 arcsec to x2cross.

These flag bits in iflag can be useful:

```
SEFLG_TRUEPOS
SEFLG_NONUT
SEFLG_EQUATORIAL (x2cross is a rectascension value, a point on the
equator, and not on the ecliptic)
```

4.7 swe_mooncross_node() and swe_mooncross_node_ut()

These functions find the crossing of the Moon over its true node, i.e. crossing through the ecliptic.

```
double swe_mooncross_node_ut(
  double tjd_ut,    // start time for search
  int32 iflag,      // flag bits
  double *xlon,     // return value, longitude at crossing time
  double *xlat,     // return value, latitude at crossing time, very near zero
  char *serr
);

double swe_mooncross_node(
  double tjd_et,    // start time for search
  int32 iflag,      // flag bits
  double *xlon,     // return value, longitude at crossing time
  double *xlat,     // return value, latitude at crossing time, very near zero
  char *serr
);
```

Return value: double jx = time of next crossing, in Ephemeris Time or Universal Time.

In case of error, a value of jx < tjd is returned. Because the crossing search is always forward in time, returning an earlier time is an indication of error. In addition, string serr will contain error details (unless it is a NULL pointer)

The position of the Moon at the moment of crossing is returned in xlon and xlat, with xlat very close to zero.

4.8 swe_helio_cross() and swe_helio_cross_ut()

There are currently no functions for geocentric crossings of other planets. Their movement is more complex because they can become stationary and retrograde and make multiple crossings in the short period of time.

There are however functions for heliocentric crossings over a position x2cross:

```
int32 swe_helio_cross_ut(
  int32 ipl,
  double x2cross,
  double tjd_ut,
  int32 iflag,
  int32 dir,
  double *jx,
  char *serr
);

int32 swe_helio_cross(
  int32 ipl,
  double x2cross,
  double tjd_et,
  int32 iflag,
  int32 dir,
  double *jx,
  char *serr
);
```

ipl is the planet number. Only objects which have a heliocentric orbit are possible.

dir >= 0 indicates search forward in time, dir < 0 indicates search backward in time. It is recommended to use dir = 1 or dir = -1.

Return value < 0 indicates an error, with error details in string serr (unless serr is a NULL pointer).

The crossing time is returned via parameter jx.

5. Fixed stars functions

The following functions are used to calculate positions of fixed stars.

5.1. Different functions for calculating fixed star positions

The function **swe_fixstar_ut()** does exactly the same as **swe_fixstar()** except that it expects Universal Time rather than Terrestrial Time (Ephemeris Time) as an input value. (cf. **swe_calc_ut()** and **swe_calc()**) For more details, see under 4.2 swe_fixstar().

In the same way, the function **swe_fixstar2_ut()** does the same as **swe_fixstar2()** except that it expects Universal Time as input time.

The functions **swe_fixstar2_ut()** and **swe_fixstar2()** were introduced with SE 2.07. They do the same as **swe_fixstar_ut()** and **swe_fixstar()** except that they are a lot faster and have a slightly different behavior, explained below.

For new projects, we recommend using the new functions **swe_fixstar2_ut()** and **swe_fixstar2()**. Performance will be a lot better *if a great number of fixed star calculations are done.* If performance is a problem with your old projects, we recommend replacing the old functions by the new ones. However, the output should be checked carefully, because the behavior of old and new functions is not exactly identical. (explained below)

5.2. swe_fixstar2_ut(), swe_fixstar2(), swe_fixstar_ut(), swe_fixstar()

```
// positions of fixed stars from UT, faster function if many stars are calculated

int32 swe_fixstar2_ut(
  char *star,     // star name and returned star name 40 bytes
  double tjd_ut,  // Julian day number, Universal Time
  int32 iflag,    // flag bits
  double *xx,     // tarray of 6 doubles
  char *serr      // 256 bytes for error string
);

// positions of fixed stars from TT, faster function if many stars are calculated

int32 swe_fixstar2(
  char *star,     // star name and returned star name 40 bytes
  double tjd_et,  // Julian day number, Ephemeris Time
  int32 iflag,    // flag bits
  double *xx,     // tarray of 6 doubles
  char *serr      // 256 bytes for error string
```

```
);

// positions of fixed stars from UT, old function

int32 swe_fixstar_ut(
  char *star,     // star name and returned star name 40 bytes
  double tjd_ut,  // Julian day number, Universal Time
  int32 iflag,    // flag bits
  double *xx,     // tarray of 6 doubles
  char *serr      // 256 bytes for error string
);

// positions of fixed stars from TT, old function

int32 swe_fixstar(
  char *star,     // star name and returned star name 40 bytes
  double tjd_et,  // Julian day number, Ephemeris Time
  int32 iflag,    // flag bits
  double *xx,     // tarray of 6 doubles
  char *serr      // 256 bytes for error string
);
```

The fixed stars functions only work if the fixed stars data file sefstars.txt is found in the ephemeris path. If the file sefstars.txt is not found, the old file fixstars.cat is searched and used instead, if present. However, **it is strongly recommended to not** use the old file anymore. The data in the file are outdated, and the algorithms are also not as accurate as those used with the file sefstars.txt.

The parameter star must provide for at least 41 characters for the returned star name. If a star is found, its name is returned in this field in the following format:
traditional_name, nomenclature_name e.g. "Aldebaran,alTau".

The nomenclature name is usually the so-called Bayer designation or the Flamsteed designation, in some cases also Henry Draper (HD) or other designations.

As for the explanation of the other parameters, see **swe_calc().**

Barycentric positions are not implemented. The difference between geocentric and heliocentric fix star position is noticeable and arises from parallax and gravitational deflection.

The function has three modes to search for a star in the file sefstars.txt:

Behavior of new functions **swe_fixstar2()** and **swe_fixstar2_ut()**:

- star contains a traditional name: the first star in the file sefstars.txt is used whose traditional name fits the given name. All names are mapped to lower case before comparison and white spaces are removed.

Changed behavior: The search string must match the complete star name. If you want to use a partial string, you have to add the wildcard character '%' to the search string, e.g. "aldeb%". (The old functions treat each search string as ending with a wildcard.)

The '%' can only be used at the end of the search string and only with the traditional star name, not with nomenclature names (i.e. not with Bayer or Flamsteed designations).

Note that the function overwrites the variable star. Both the full traditional name and the nomenclature name are copied into the variable, separated by a comma. E.g. if star is given the value "aldeb", then swe_fixstar() overwrites this with "Aldebaran,alTau". The new string can also be used for a new search of the same star.

- star contains a comma, followed by a nomenclature name, e.g. ",alTau": the search string is understood to be the nomenclature name (the second field in a star record). Letter case is observed in the comparison for nomenclature names.

- star contains a positive number (in ASCII string format, e.g. ”234”):

Changed behavior: The numbering of stars follows a sorted list of nomenclature names. (With the old functions, the n-th star of the fixed star file is returned.)

Behavior of old functions **swe_fixstar()** and **swe_fixstar_ut()**:

- star contains a traditional name: the first star in the file sefstars.txt is used whose traditional name fits the given name. All names are mapped to lower case before comparison and white spaces are removed.

If star has n characters, only the first n characters of the traditional name field are compared.

Note that the function overwrites the variable star. Both the full traditional name and the nomenclature name are copied into the variable, separated by a comma. E.g. if star is given the value “aldeb”, then **swe_fixstar()** overwrites this with “Aldebaran,alTau”. The new string can also be used for a new search of the same star.

- star begins with a comma, followed by a nomenclature name, e.g. ”,alTau”: the search string is understood to be the nomenclature name (the second field in a star record). Letter case is observed in the comparison for nomenclature names. Here again, star is overwritten by the string “Aldebaran,alTau”.
- star contains a positive number (in ASCII string format, e.g. ”234”):

The star data in the 234-th non-comment line in the file sefstars.txt are used. Comment lines that begin with # and are ignored. Here again, star will be overwritten by the traditional name and the nomenclature name, separated by a comma, e.g. “Aldebaran,alTau”.

For correct spelling of nomenclature names, see file sefstars.txt. Nomenclature names are usually Bayer designations and are composed of a Greek letter and the name of a star constellation. The Greek letters were originally used to write numbers, therefore they actually number the stars of the constellation. The abbreviated nomenclature names we use in sefstars.txt are constructed from two lowercase letters for the Greek letter (e.g. “al” for “alpha”, except “omi” and “ome”) and three letters for the constellation (e.g. “Tau” for “Tauri”).

The searching of stars by sequential number (instead of name or nomenclature name) is a practical feature if one wants to list all stars:

```
for i=1; i<10000; i++) { // choose any number greater than number of lines (stars) in file
  sprintf(star, "%d", i);
  returncode = swe_fixstar2(star, tjd, ...);
  // ... whatever you want to do with the star positions ...
  if (returncode == ERR)
    break;
}
```

The function should survive damaged sefstars.txt files which contain illegal data and star names exceeding the accepted length. Such fields are cut to acceptable length.

There are a few special entries in the file sefstars.txt:

```
# Gal. Center (SgrA*) according to Simbad database,
# speed of SgrA* according to Reid (2004), "The Proper Motion of Sagittarius A*",
# p. 873: -3.151 +- 0.018 mas/yr, -5.547 +- 0.026 mas/yr. Component in RA must be
# multiplied with cos(decl).
Galactic Center,SgrA*,ICRS,17,45,40.03599,-29,00,28.1699,-2.755718425,-5.547, 0.0,0.125,999.99, 0, 0
# Great Attractor, near Galaxy Cluster ACO 3627, at gal. coordinates
# 325.3, -7.2, 4844 km s-1 according to Kraan-Korteweg et al. 1996, Woudt 1998
Great Attractor,GA,2000,16,15,02.836,-60,53,22.54,0.000,0.00,0.0,0.0000159,999.99, 0, 0
# Virgo Cluster, according to NED (Nasa Extragalactic Database)
Virgo Cluster,VC,2000,12,26,32.1,12,43,24,0.000, 0.00, 0.0,0.0000,999.99, 0, 0
# The solar apex, or the Apex of the Sun's Way, refers to the direction that the Sun travels
# with respect to the so-called Local Standard of Rest.
```

```
Apex ,Apex,1950,18,03,50.2, 30,00,16.8, 0.000, 0.00,-16.5,0.0000,999.99, 0, 0
# Galactic Pole acc. to Liu/Zhu/Zhang, „Reconsidering the galactic coordinate system",
# Astronomy & Astrophysics No. AA2010, Oct. 2010, p. 8.
# It is defined relative to a plane that contains the galactic center and the Sun and
# approximates the galactic plane.
Gal.Pole,GPol,ICRS,12,51,36.7151981,27,06,11.193172,0.0,0.0,0.0,0.0,0.0,0,0
# Old Galactic Pole IAU 1958 relative to ICRS according to the same publication p. 7
Gal.Pole IAU1958,GP1958,ICRS,12,51,26.27469,27,07,41.7087,0.0,0.0,0.0,0.0,0.0,0,0
# Old Galactic Pole relative to ICRS according to the same publication p. 7
Gal.Pole IAU1958,GP1958,ICRS,12,51,26.27469,27,07,41.7087,0.0,0.0,0.0,0.0,0.0,0,0
# Pole of true galactic plane, calculated by DK
Gal.Plane Pole,GPPlan,ICRS,12,51,5.731104,27,10,39.554849,0.0,0.0,0.0,0.0,0.0,0,0
# The following "object" played an important role in 2011 and 2017 dooms day predictions,
# as well as in some conspiration theories. It consists of the infrared objects
# IRAS 13458-0823 and IRAS 13459-0812. Central point measured by DK.
Infrared Dragon,IDrag, ICRS,13,48,0.0,-9,0,0.0,0,0,0,0,0.0, 19, 477
```

You may edit the star catalogue and move the stars you prefer to the top of the file. With older versions of the Swiss Ephemeris, this will increase the speed of computations. The search mode is linear through the whole star file for each call of **swe_fixstar()**.

However, with the new functions **swe_fixstar2()** and **swe_fixstar2_ut()**, calculation speed will be the same for all stars.

Attention:

With older versions of the Swiss Ephemeris, **swe_fixstar() does not compute speeds** of the fixed stars. Also, distance is always returned as 1 for all stars. Since SE 2.07 distances and daily motions are included in the return array, if SEFLG_SPEED is set in parameter iflag.

Distances are given in AU. To convert them from AU to lightyears or parsec, please use the following defines, which are located in swephexp.h:

```
#define SE_AUNIT_TO_LIGHTYEAR (1.0/63241.077088071)

#define SE_AUNIT_TO_PARSEC (1.0/206264.8062471)
```

The daily motions of the fixed stars contain components of precession, nutation, aberration, parallax and the proper motions of the stars.

5.3. swe_fixstar2_mag(), swe_fixstar_mag()

```
int32 swe_fixstar2_mag(
  char *star,
  double *mag,
  char *serr
);

int32 swe_fixstar_mag(      // old function
  char *star,
  double *mag,
  char *serr
);
```

This function calculates the magnitude of a fixed star. The function returns OK or ERR. The magnitude value is returned in the parameter mag.

For the definition and use of the parameter star see function **swe_fixstar()**. The parameter serr and is, as usually, an error string pointer.

The new function **swe_fixstar2_mag()** (since SE 2.07) is more efficient if great numbers of fixed stars are calculated.

Strictly speaking, the magnitudes returned by this function are valid for the year 2000 only. Variations in brightness due to the star's variability or due to the increase or decrease of the star's distance cannot be taken into account. With stars of constant absolute magnitude, the change in brightness can be ignored for the historical period. E.g. the current magnitude of Sirius is -1.46. In 3000 BCE it was -1.44.

6. Planetary risings, settings, meridian transits, planetary phenomena

6.1. swe_rise_trans() and swe_rise_trans_true_hor() (risings, settings, meridian transits)

The function **swe_rise_trans()** computes the times of rising, setting and meridian transits for all planets, asteroids, the moon, and the fixed stars. The function **swe_rise_trans_true_hor()** does the same for a local horizon that has an altitude != 0.

The function returns a rising time of an object:

- if at t0 the object is below the horizon and a rising takes place before the next culmination of the object;
- if at t0 the object is above the horizon and a rising takes place between the next lower and upper culminations of the object.

And it returns a setting time of an object,

- if at t0 the object is above the horizon and a setting takes place before the next lower culmination of the object;
- if at t0 the object is below the horizon and a setting takes place between the next upper and lower culminations.

Note, "culmination" does not mean meridian transit, especially not with the Sun, Moon, and planets. The culmination of a moving body with changing declination does not take place exactly on the meridian but shortly before or after the meridian transit. In polar regions, it even happens that the moon "rises" shortly after the culmination, on the west side of the meridian. I. e., the upper limb if its disk will become visible for a short time. The function **swe_rise_trans()** should catch these cases.

Function definitions are as follows:

```
int32 swe_rise_trans(
  double tjd_ut,    // search after this time (UT)
  int32 ipl,        // planet number, if planet or moon
  char *starname,   // star name, if star; must be NULL or empty, if ipl > is used
  int32 epheflag,   // ephemeris flag
  int32 rsmi,       // integer specifying that rise, set, or one of the two
           // meridian transits is wanted. see definition below
  double *geopos,   // array of three doubles containing
           // geograph. long., lat., height of observer
  double atpress    // atmospheric pressure in mbar/hPa
```

```
  double attemp,    // atmospheric temperature in deg. C
  double *tret,     // return address (double) for rise time etc.
  char *serr        // return address for error message
);

int32 swe_rise_trans_true_hor(
  double tjd_ut,    // search after this time (UT)
  int32 ipl,        // planet number, if planet or moon
  char *starname,   // star name, if star; must be NULL or empty, if ipl > is used
  int32 epheflag,   // ephemeris flag
  int32 rsmi,       // integer specifying that rise, set, or one of the two
            // meridian transits is wanted. see definition below
  double *geopos,   // array of three doubles containing
            // geograph. long., lat., height of observer
  double atpress    // atmospheric pressure in mbar/hPa
  double attemp,    // atmospheric temperature in deg. C
  double horhgt,    // height of local horizon in deg at the point where
            // the body rises or sets
  double *tret,     // return address (double) for rise time etc.
  char *serr        // return address for error message
);
```

The second function has one additional parameter horhgt for the height of the local horizon at the point where the body rises or sets.

The variable rsmi can have the following values:

```
#define SE_CALC_RISE 1
#define SE_CALC_SET 2
#define SE_CALC_MTRANSIT 4 // upper meridian transit (southern for northern geo. latitudes)
#define SE_CALC_ITRANSIT 8 // lower meridian transit (northern, below the horizon)
    //  the following bits can be added (or'ed) to SE_CALC_RISE or SE_CALC_SET
#define SE_BIT_DISC_CENTER 256 // for rising or setting of disc center.
#define SE_BIT_DISC_BOTTOM 8192 // for rising or setting of lower limb of disc
#define SE_BIT_GEOCTR_NO_ECL_LAT 128 // use topocentric position of object and ignore its ecliptic lati
#define SE_BIT_NO_REFRACTION 512 // if refraction is not to be considered
#define SE_BIT_CIVIL_TWILIGHT 1024 // in order to calculate civil twilight
#define SE_BIT_NAUTIC_TWILIGHT 2048 // in order to calculate nautical twilight
#define SE_BIT_ASTRO_TWILIGHT 4096 // in order to calculate astronomical twilight
#define SE_BIT_FIXED_DISC_SIZE (16*1024) // neglect the effect of distance on disc size
#define SE_BIT_HINDU_RISING (SE_BIT_DISC_CENTER | SE_BIT_NO_REFRACTION | SE_BIT_GEOCTR_NO_ECL_LAT)
    // risings according to Hindu astrology
```

rsmi = 0 will return risings.

The rising times depend on the atmospheric pressure and temperature. atpress expects the atmospheric pressure in millibar (hectopascal); attemp the temperature in degrees Celsius.

If atpress is given the value 0, the function estimates the pressure from the geographical altitude given in geopos[2] and attemp. If geopos[2] is 0, atpress will be estimated for sea level.

Function return values are:

- 0 if a rising, setting or transit event was found;
- -1 if an error occurred (usually an ephemeris problem);
- -2 if a rising or setting event was not found because the object is circumpolar.

Sunrise in Astronomy and in Hindu Astrology

The astronomical sunrise is defined as the time when the upper limb of the solar disk is seen appearing at the horizon. The astronomical sunset is defined as the moment the upper limb of the solar disk disappears below the horizon.

The function **swe_rise_trans()** by default follows this definition of astronomical sunrises and sunsets. Also, astronomical almanacs and newspapers publish astronomical sunrises and sunset according to this definition.

Hindu astrology and Hindu calendars use a different definition of sunrise and sunset. They consider the Sun as rising or setting, when the center of the solar disk is exactly at the horizon. In addition, the Hindu method ignores atmospheric refraction. Moreover, the geocentric rather than topocentric position is used and the small ecliptic latitude of the Sun is ignored.

In order to calculate correct Hindu rising and setting times, the flags SE_BIT_NO_REFRACTION and SE_BIT_DISC_CENTER must be added (or'ed) to the parameter rsmi. From Swiss Ephemeris version 2.06 on, a flag SE_BIT_HINDU_RISING is supported. It includes the flags SE_BIT_NO_REFRACTION, SE_BIT_DISC_CENTER and SE_BIT_GEOCTR_NO_ECL_LAT.

In order to calculate the sunrise of a given date and geographic location, one can proceed as in the following program (tested code!):

```
int main()
{
  char serr[AS_MAXCH];
  double epheflag = SEFLG_SWIEPH;
  int gregflag = SE_GREG_CAL;
  int year = 2017;
  int month = 4;
  int day = 12;
  int geo_longitude = 76.5; // positive for east, negative for west of Greenwich
  int geo_latitude = 30.0;
  int geo_altitude = 0.0;
  double hour;
  // array for atmospheric conditions
  double datm[2];
  datm[0] = 1013.25; // atmospheric pressure;
  // irrelevant with Hindu method, can be set to 0
  datm[1] = 15; // atmospheric temperature;
  // irrelevant with Hindu method, can be set to 0
  // array for geographic position
  double geopos[3];
  geopos[0] = geo_longitude;
  geopos[1] = geo_latitude;
  geopos[2] = geo_altitude; // height above sea level in meters;
  // irrelevant with Hindu method, can be set to 0
  swe_set_topo(geopos[0], geopos[1], geopos[2]);
  int ipl = SE_SUN; // object whose rising is wanted
  char starname[255]; // name of star, if a star's rising is wanted
  // is "" or NULL, if Sun, Moon, or planet is calculated
  double trise; // for rising time
  double tset; // for setting time
  // calculate the Julian day number of the date at 0:00 UT:
  double tjd = swe_julday(year,month,day,0,gregflag);
  // convert geographic longitude to time (day fraction) and subtract it from tjd
  // this method should be good for all geographic latitudes except near in
  // polar regions
```

```
  double dt = geo_longitude / 360.0;
  tjd = tjd - dt;
  // calculation flag for Hindu risings/settings
  int rsmi = SE_CALC_RISE | SE_BIT_HINDU_RISING;
  // or SE_CALC_RISE + SE_BIT_HINDU_RISING;
  // or SE_CALC_RISE | SE_BIT_DISC_CENTER | SE_BIT_NO_REFRACTION | SE_BIT_GEOCTR_NO_ECL_LAT;
  int return_code = swe_rise_trans(tjd, ipl, starname, epheflag, rsmi, geopos, datm[0], datm[1], &trise,
  if (return_code == ERR) { // error action
    printf("%s\n", serr);
    returnn ERR;
  }
  // conversion to local time zone must be made by the user. The Swiss Ephemeris
  // does not have a function for that.
  // After that, the Julian day number of the rising time can be converted into
  // date and time:
  swe_revjul(trise, gregflag, &year, &month, &day, &hour);
  printf("sunrise: date=%d/%d/%d, hour=%.6f UT\n", year, month, day, hour);
  // To calculate the time of the sunset, you can either use the same
  // tjd increased or trise as start date for the search.
  rsmi = SE_CALC_SET | SE_BIT_DISC_CENTER | SE_BIT_NO_REFRACTION;
  return_code = swe_rise_trans(tjd, ipl, starname, epheflag, rsmi, geopos, datm[0], datm[1], &tset, serr
  if (return_code == ERR) { // error action
    printf("%s\n", serr);
    return ERR;
  }
  printf("sunset : date=%d/%d/%d, hour=%.6f UT\n", year, month, day, hour);
}
```

6.2. swe_pheno_ut() and swe_pheno(), planetary phenomena

These functions compute phase, phase angle, elongation, apparent diameter, apparent magnitude for the Sun, the Moon, all planets and asteroids. The two functions do exactly the same but expect a different time parameter.

```
int32 swe_pheno_ut(
  double tjd_ut,// time Jul. Day UT
  int32 ipl,    // planet number
  int32 iflag,  // ephemeris flag
  double *attr, // return array, 20 doubles, see below
  char *serr    // return error string
);

int32 swe_pheno(
  double tjd_et,// time Jul. Day ET
  int32 ipl,    // planet number
  int32 iflag,  // ephemeris flag
  double *attr, // return array, 20 doubles, see below
  char *serr    // return error string
);
```

The function returns:

```
  attr[0] = phase angle (Earth-planet-sun)
  attr[1] = phase (illumined fraction of disc)
```

```
attr[2] = elongation of planet
attr[3] = apparent diameter of disc
attr[4] = apparent magnitude
```

declare as attr[20] at least!

NOTE: the lunar magnitude is quite a complicated thing, but our algorithm is very simple. The phase of the moon, its distance from the Earth and the sun is considered, but no other factors.

iflag also allows SEFLG_TRUEPOS, SEFLG_HELCTR

6.3. swe_azalt(), horizontal coordinates, azimuth, altitude

swe_azalt() computes the horizontal coordinates (azimuth and altitude) of a planet or a star from either ecliptical or equatorial coordinates.

```
#define SE_ECL2HOR 0
#define SE_EQU2HOR 1

void swe_azalt*(
  double tjd_ut,  // UT
  int32 calc_flag,    // SE_ECL2HOR or SE_EQU2HOR
  double *geopos, // array of 3 doubles: geograph. long., lat., height
  double atpress, // atmospheric pressure in mbar (hPa)
  double attemp,  // atmospheric temperature in degrees Celsius
  double *xin,    // array of 3 doubles: position of body in either ecliptical
         // or equatorial coordinates, depending on calc_flag
  double *xaz     // return array of 3 doubles, containing azimuth, true altitude, apparent altitude
);
```

If **calc_flag** = SE_ECL2HOR, set

```
xin[0] = ecl. long.,
xin[1] = ecl. lat.,
(xin[2] = distance (not required));
```

else

if **calc_flag** = SE_EQU2HOR, set

```
xin[0] = right ascension,
xin[1] = declination,
(xin[2] = distance (not required))
```

The return values are:

- xaz[0] = azimuth, i.e. position degree, measured from the south point to west;
- xaz[1] = true altitude above horizon in degrees;
- xaz[2] = apparent (refracted) altitude above horizon in degrees.

The apparent altitude of a body depends on the atmospheric pressure and temperature. If only the true altitude is required, these parameters can be neglected.

If atpress is given the value 0, the function estimates the pressure from the geographical altitude given in geopos[2] and attemp. If geopos[2] is 0, atpress will be estimated for sea level.

6.4. swe_azalt_rev()

The function **swe_azalt_rev()** is not precisely the reverse of **swe_azalt()**. It computes either ecliptical or equatorial coordinates from azimuth and true altitude. If only an apparent altitude is given, the true altitude has to be computed first with the function **swe_refrac()** (see below).

It is defined as follows:

```
void swe_azalt_rev(
  double tjd_ut,
  int32 calc_flag,  // either SE_HOR2ECL or SE_HOR2EQU
  double *geopos,   // array of 3 doubles for geograph. pos. of observer
  double *xin,      // array of 2 doubles for azimuth and true altitude of > planet
  double *xout      // return array of 2 doubles for either ecliptic or
            // equatorial coordinates, depending on calc_flag
);
```

6.5. swe_refrac(), swe_refrac_extended(), refraction

The refraction function **swe_refrac()** calculates either the true altitude from the apparent altitude or the apparent altitude from the apparent altitude. Its definition is:

```
double swe_refrac(
  double inalt,
  double atpress,   // atmospheric pressure in mbar (hPa) */
  double attemp,    // atmospheric temperature in degrees Celsius */
  int32 calc_flag   // either SE_TRUE_TO_APP or SE_APP_TO_TRUE */
);
```

where:

```
#define SE_TRUE_TO_APP 0
#define SE_APP_TO_TRUE 1
```

The refraction depends on the atmospheric pressure and temperature at the location of the observer.

If atpress is given the value 0, the function estimates the pressure from the geographical altitude given in geopos[2] and attemp**.** If geopos[2] is 0, atpress will be estimated for sea level.

There is also a more sophisticated function **swe_refrac_extended()**. It allows correct calculation of refraction for altitudes above sea > 0, where the ideal horizon and planets that are visible may have a negative height. (for swe_refrac(), negative apparent heights do not exist!)

```
double swe_refrac_extended(
  double inalt,     // altitude of object above geometric horizon in degrees,
            // where geometric horizon = plane perpendicular to gravity
  double geoalt,    // altitude of observer above sea level in meters
  double atpress,   // atmospheric pressure in mbar (hPa)
  double lapse_rate,    // (dattemp/dgeoalt) = [°K/m]
  double attemp,    // atmospheric temperature in degrees Celsius
  int32 calc_flag,  // either SE_TRUE_TO_APP or SE_APP_TO_TRUE
  double *dret      // array of 4 doubles; can be NULL
);
```

- dret[0] true altitude, if possible; otherwise input value
- dret[1] apparent altitude, if possible; otherwise input value

- dret[2] refraction
- dret[3] dip of the horizon

The function returns:

- **case SE_TRUE_TO_APP**, conversion from true altitude to apparent altitude:
  - apparent altitude, if body appears above is observable above ideal horizon;
  - true altitude (the input value); otherwise "ideal horizon" is the horizon as seen above an ideal sphere (as seen from a plane over the ocean with a clear sky)
- **case SE_APP_TO_TRUE**, conversion from apparent altitude to true altitude:
  - the true altitude resulting from the input apparent altitude, if this value is a plausible apparent altitude, i.e. if it is a position above the ideal horizon;
  - the input altitude; otherwise in addition the array dret[] returns the following values, if not a NULLL pointer:
    * dret[0] true altitude, if possible; otherwise input value;
    * dret[1] apparent altitude, if possible; otherwise input value;
    * dret[2] refraction;
    * dret[3] dip of the horizon.

The body is above the horizon if the dret[0] != dret[1].

6.6. Heliacal risings etc.: swe_heliacal_ut()

The function **swe_heliacal_ut()** the Julian day of the next heliacal phenomenon after a given start date. It works between geographic latitudes 60s – 60n.

```
int32 swe_heliacal_ut(
  double tjdstart,      // Julian day number of start date for the search of the heliacal event
  double *dgeo          // geographic position (details below)
  double *datm,     // atmospheric conditions (details below)
  double *dobs,     //  observer description (details below)
  char *objectname,     //  name string of fixed star or planet
  int32 event_type,     //  event type (details below)
  int32 helflag,    //  calculation flag, bitmap (details below)
  double *dret,     //  result: array of at least 50 doubles, of which 3 are used at the moment
  char * serr       // error string
);
```

Function returns OK or ERR.

Input values:

```
Details for dgeo[] (array of doubles):
  dgeo[0]: geographic longitude;
  dgeo[1]: geographic latitude;
  dgeo[2]: geographic altitude (eye height) in meters.

Details for datm[] (array of doubles):
  datm[0]: atmospheric pressure in mbar (hPa) ;
  datm[1]: atmospheric temperature in degrees Celsius;
  datm[2]: relative humidity in %;
  datm[3]: if datm[3]>=1, then it is Meteorological Range [km] ;
```

```
if 1> datm[3] >0, then it is the total atmospheric coefficient (ktot) ;

if datm[3]=0, then the other atmospheric parameters determine the total
atmospheric coefficient (ktot)
```

Default values:

If this is too much for you, set all these values to 0. The software will then set the following defaults:

Pressure 1013.25, temperature 15, relative humidity 40. The values will be modified depending on the altitude of the observer above sea level.

If the extinction coefficient (meteorological range) datm[3] is 0, the software will calculate its value from datm[0..2].

```
Details for dobs[] (array of six doubles):
  dobs[0]: age of observer in years (default = 36)
  dobs[1]: Snellen ratio of observers eyes (default = 1 = normal)

The following parameters are only relevant if the flag SE_HELFLAG_OPTICAL_PARAMS is set:
  dobs[2]: 0 = monocular, 1 = binocular (actually a boolean)
  dobs[3]: telescope magnification: 0 = default to naked eye (binocular), 1 = naked eye
  dobs[4]: optical aperture (telescope diameter) in mm
  dobs[5]: optical transmission

Details for event_type:
 event_type = SE_HELIACAL_RISING (1): morning first (exists for all visible planets and stars);
 event_type = SE_HELIACAL_SETTING (2): evening last (exists for all visible planets and stars);
  event_type = SE_EVENING_FIRST (3): evening first (exists for Mercury, Venus, and the Moon);
  event_type = SE_MORNING_LAST (4): morning last (exists for Mercury, Venus, and the Moon).
```

Details for helflag:

helflag contains ephemeris flag, like iflag in **swe_calc()** etc. In addition it can contain the following bits:

- SE_HELFLAG_OPTICAL_PARAMS (512): Use this with calculations for optical instruments.

Unless this bit is set, the values of dobs[2-5] are ignored.

- SE_HELFLAG_NO_DETAILS (1024): provide the date, but not details like visibility start, optimum, and end. This bit makes the program a bit faster.
- SE_HELFLAG_VISLIM_DARK (4096): function behaves as if the Sun were at nadir.
- SE_HELFLAG_VISLIM_NOMOON (8192): function behaves as if the Moon were at nadir, i. e. the Moon as a factor disturbing the observation is excluded. This flag is useful if one is not really interested in the heliacal date of that particular year, but in the heliacal date of that epoch.

Some other SE_HELFLAG_ bits found in swephexp.h were made for mere test purposes and may change in future releases. Please **do not use them** and do not request any support or information related to them.

```
Details for return array dret[] (array of doubles):
  dret[0]: start visibility (Julian day number);
  dret[1]: optimum visibility (Julian day number), zero if helflag >= SE_HELFLAG_AV;
  dret[2]: end of visibility (Julian day number), zero if helflag >= SE_HELFLAG_AV.
```

Strange phenomena:

- Venus' heliacal rising can occur before her heliacal setting. In such cases the planet may be seen both as a morning star and an evening star for a couple of days. Example:

swetest -hev1 -p3 -b1.1.2008 -geopos8,47,900 -at1000,10,20,0.15 -obs21,1 -n1 -lmt

Venus heliacal rising : 2009/03/23 05:30:12.4 LMT (2454913.729310), visible for: 4.9 min

swetest -hev2 -p3 -b1.1.2008 -geopos8,47,900 -at1000,10,20,0.15 -obs21,1 -n1 -lmt

Venus heliacal setting: 2009/03/25 18:37:41.6 LMT (2454916.276175), visible for: 15.1 min

- With good visibility and good eye sight (high Snellen ratio), the "evening first" of the Moon may actually begin in the morning, because the Moon becomes visible before sunset. Note the LMT and duration of visibility in the following example:

swetest -hev3 -p1 -b1.4.2008 -geopos8,47,900 -at1000,10,40,0.15 -obs21,1.5 -n1 -lmt

Moon evening first : 2008/04/06 10:33:44.3 LMT (2454562.940096), visible for: 530.6 min

- Stars that are circumpolar, but come close to the horizon, may have an evening last and a morning first, but **swe_heliacal_ut()** will not find it. It only works if a star crosses the horizon.
- In high geographic latitudes > 55 (?), unusual things may happen. E.g. Mars can have a morning last appearance. In case the period of visibility lasts for less than 5 days, the function **swe_heliacal_ut()** may miss the morning first.
- With high geographic latitudes heliacal appearances of Mercury and Venus become rarer.

The user must be aware that strange phenomena occur especially for high geographic latitudes and circumpolar objects and that the function **swe_heliacal_ut()** may not always be able to handle them correctly. Special cases can best be researched using the function **swe_vis_limit_mag()**.

6.7. Magnitude limit for visibility: swe_vis_limit_mag()

The function **swe_vis_limit_mag()** determines the limiting visual magnitude in dark skies. If the visual magnitude mag of an object is known for a given date (e. g. from a call of function **swe_pheno_ut()**, and if mag is smaller than the value returned by **swe_vis_limit_mag()**, then it is visible.

Please note that this is an unusal SE function. Instead of a planet number it requests the name of the object.

```
double swe_vis_limit_mag(
  double tjdut, // Julian day number
  double *dgeo  // geographic position (details under swe_heliacal_ut()
  double *datm, // atmospheric conditions (details under swe_heliacal_ut())
  double *dobs, // observer description (details under swe_heliacal_ut())
  char *objectname,     // name string of fixed star or planet
  int32 helflag,    // calculation flag, bitmap (details under swe_heliacal_ut())
  double *dret,     // result: array of 8 doubles
  char *serr    // error string
);
```

Function returns:

- -1 on error;
- -2 object is below horizon;
- 0 , photopic vision;
- 1 , scotopic vision;
- 2 , photopic, but near limit photopic/scotopic vision.
- 3 , scotopic, but near limit photopic/scotopic vision.

Details for arrays dgeo[], datm[], dobs[] and the other input parameters are given under "7.17. Heliacal risings etc.: **swe_heliacal_ut()**".

```
Details for return array dret[] (array of doubles):
 dret[0]: limiting visual magnitude (if dret[0] > magnitude of object, then the object is visible);
  dret[1]: altitude of object;
  dret[2]: azimuth of object;
```

```
  dret[3]: altitude of sun;
  dret[4]: azimuth of sun;
  dret[5]: altitude of moon;
  dret[6]: azimuth of moon;
  dret[7]: magnitude of object.
```

6.8. Heliacal details: swe_heliacal_pheno_ut()

The function **swe_heliacal_pheno_ut()** provides data that are relevant for the calculation of heliacal risings and settings. This function does not provide data of heliacal risings and settings, just some additional data mostly used for test purposes. To calculate heliacal risings and settings, please use the function **swe_heliacal_ut()** documented further above.

```
double swe_heliacal_pheno_ut(
  double tjd_ut,    // Julian day number
  double *dgeo,     // geographic position (details under swe_heliacal_ut()
  double *datm,     // atmospheric conditions (details under swe_heliacal_ut())
  double *dobs,     // observer description (details under swe_heliacal_ut())
  char *objectname,     // name string of fixed star or planet
  int32 event_type,     // event type (details under function swe_heliacal_ut())
  int32 helflag,    // calculation flag, bitmap (details under swe_heliacal_ut())
  double *darr,     // return array, declare array of 50 doubles
  char *serr        // error string
);

The return array has the following data:
  darr[0] =  AltO [deg] topocentric altitude of object (unrefracted)
  darr[1] = AppAltO [deg] apparent altitude of object (refracted)
  darr[2] = GeoAltO [deg] geocentric altitude of object
  darr[3] = AziO [deg] azimuth of object
  darr[4] = AltS [deg] topocentric altitude of Sun
  darr[5] = AziS [deg] azimuth of Sun
  darr[6] = TAVact [deg] actual topocentric arcus visionis
  darr[7] = ARCVact [deg] actual (geocentric) arcus visionis
  darr[8] = DAZact [deg] actual difference between object's and sun's azimuth
  darr[9] = ARCLact [deg] actual longitude difference between object and sun
  darr[10] = kact [-] extinction coefficient
  darr[11] = minTAV [deg] smallest topocentric arcus visionis
  darr[12] = TfistVR [JDN] first time object is visible, according to VR
  darr[13] = TbVR [JDN optimum time the object is visible, according to VR
  darr[14] = TlastVR [JDN] last time object is visible, according to VR
  darr[15] = TbYallop [JDN] best time the object is visible, according to Yallop
  darr[16] = WMoon [deg] crescent width of Moon
  darr[17] = qYal [-] q-test value of Yallop
  darr[18] = qCrit [-] q-test criterion of Yallop
  darr[19] = ParO [deg] parallax of object
  darr[20] = Magn [-] magnitude of object
  darr[21] = RiseO [JDN] rise/set time of object
  darr[22] = RiseS [JDN] rise/set time of Sun
  darr[23] = Lag [JDN] rise/set time of object minus rise/set time of Sun
  darr[24] = TvisVR [JDN] visibility duration
  darr[25] = LMoon [deg] crescent length of Moon
  darr[26] = CVAact [deg]
```

```
darr[27] = Illum [%] new
darr[28] = CVAact [deg] new
darr[29] = MSk [-]
```

7. Eclipses and Occultations

There are the following functions for eclipse and occultation calculations.

Solar eclipses:

- **swe_sol_eclipse_when_loc**(tjd...) finds the next eclipse for a given geographic position;
- **swe_sol_eclipse_when_glob**(tjd...) finds the next eclipse globally;
- **swe_sol_eclipse_where()** computes the geographic location of a solar eclipse for a given tjd;
- **swe_sol_eclipse_how()** computes attributes of a solar eclipse for a given tjd, geographic longitude, latitude and height.

Occultations of planets by the moon:

These functions can also be used for solar eclipses. But they are slightly less efficient.

- **swe_lun_occult_when_loc**(tjd...) finds the next occultation for a body and a given geographic position;
- **swe_lun_occult_when_glob**(tjd...) finds the next occultation of a given body globally;
- **swe_lun_occult_where()** computes the geographic location of an occultation for a given tjd.

Lunar eclipses:

- **swe_lun_eclipse_when_loc**(tjd...) finds the next lunar eclipse for a given geographic position;
- **swe_lun_eclipse_when**(tjd...) finds the next lunar eclipse;
- **swe_lun_eclipse_how()** computes the attributes of a lunar eclipse for a given tjd.

Risings, settings, and meridian transits of planets and stars:

- **swe_rise_trans()**;
- **swe_rise_trans_true_hor()** returns rising and setting times for a local horizon with altitude != 0.

Planetary phenomena:

- **swe_pheno_ut()** and **swe_pheno()** compute phase angle, phase, elongation, apparent diameter, and apparent magnitude of the Sun, the Moon, all planets and asteroids.

7.1. Example of a typical eclipse calculation

Find the next total eclipse, calculate the geographical position where it is maximal and the four contacts for that position (for a detailed explanation of all eclipse functions see the next chapters):

```
double tret[10], attr[20], geopos[10];
char serr[255];
int32 whicheph = 0; // default ephemeris
double tjd_start = 2451545; // Julian day number for 1 Jan 2000
```

```
int32 ifltype = SE_ECL_TOTAL | SE_ECL_CENTRAL | SE_ECL_NONCENTRAL;

// find next eclipse anywhere on Earth */
eclflag = swe_sol_eclipse_when_glob(tjd_start, whicheph, ifltype, tret, 0, serr);
if (eclflag == ERR)
  return ERR;
// the time of the greatest eclipse has been returned in tret[0];
// now we can find geographical position of the eclipse maximum */
tjd_start = tret[0];
eclflag = swe_sol_eclipse_where(tjd_start, whicheph, geopos, attr, serr);
if (eclflag == ERR)
  return ERR;
// the geographical position of the eclipse maximum is in geopos[0] and geopos[1];
// now we can calculate the four contacts for this place. The start time is chosen
// a day before the maximum eclipse:
tjd_start = tret[0] - 1;
eclflag = swe_sol_eclipse_when_loc(tjd_start, whicheph, geopos, tret, attr, 0, serr);
if (eclflag == ERR)
  return ERR;
// now tret[] contains the following values:
// tret[0] = time of greatest eclipse (Julian day number)
// tret[1] = first contact
// tret[2] = second contact
// tret[3] = third contact
// tret[4] = fourth contact
```

7.2. swe_sol_eclipse_when_glob()

To find the next eclipse globally:

```
int32 swe_sol_eclipse_when_glob(
    double tjd_start, // start date for search, Jul. day UT
    int32 ifl,    // ephemeris flag
    int32 ifltype,    // eclipse type wanted: SE_ECL_TOTAL etc. or 0, if any > eclipse type
    double *tret,     // return array, 10 doubles, see below
    AS_BOOL backward, // TRUE, if backward search
    char *serr    // return error string
);
```

This function requires the time parameter tjd_start in *Universal Time* and also yields the return values (tret[]) in UT. For conversions between ET and UT, use the function **swe_deltat()**.

Note: An implementation of this function with parameters in Ephemeris Time would have been possible. The question when the next solar eclipse will happen anywhere on Earth is independent of the rotational position of the Earth and therefore independent of Delta T. However, the function is often used in combination with other eclipse functions (see example below), for which input and output in ET makes no sense, because they concern local circumstances of an eclipse and therefore *are* dependent on the rotational position of the Earth. For this reason, UT has been chosen for the time parameters of all eclipse functions.

ifltype specifies the eclipse type wanted. It can be a combination of the following bits (see swephexp.h):

```
#define SE_ECL_CENTRAL 1
#define SE_ECL_NONCENTRAL 2
#define SE_ECL_TOTAL 4
#define SE_ECL_ANNULAR 8
```

```
#define SE_ECL_PARTIAL 16
#define SE_ECL_ANNULAR_TOTAL 32
```

Recommended values for ifltype:

```
// search for any eclipse, no matter which type \*/
  ifltype = 0;
// search a total eclipse; note: non-central total eclipses are very rare
  ifltype = SE_ECL_TOTAL ¦ SE_ECL_CENTRAL ¦ SE_ECL_NONCENTRAL;
// search an annular eclipse
  ifltype = SE_ECL_ANNULAR ¦ SE_ECL_CENTRAL ¦ SE_ECL_NONCENTRAL;
// search an annular-total (hybrid) eclipse
  ifltype\_ = SE_ECL_ANNULAR_TOTAL ¦ SE_ECL_CENTRAL ¦ SE_ECL_NONCENTRAL;
// search a partial eclipse
  ifltype = SE_ECL_PARTIAL;
```

If your code does not work, please study the sample code in swetest.c.

The function returns:

```
retflag = -1 (ERR) on error (e.g. if swe_calc() for sun or moon fails)
    = 0  if no eclipse
    = SE_ECL_TOTAL or SE_ECL_ANNULAR or SE_ECL_PARTIAL or SE_ECL_ANNULAR_TOTAL
    combined with
    SE_ECL_CENTRAL
    SE_ECL_NONCENTRAL

  tret[0] time of maximum eclipse
  tret[1] time, when eclipse takes place at local apparent noon
  tret[2] time of eclipse begin
  tret[3] time of eclipse end
  tret[4] time of totality begin
  tret[5] time of totality end
  tret[6] time of center line begin
  tret[7] time of center line end
  tret[8] time when annular-total eclipse becomes total, not implemented so far
  tret[9] time when annular-total eclipse becomes annular again, not implemented so far
```

declare as tret[10] at least!

7.3. swe_sol_eclipse_when_loc()

To find the next eclipse for a given geographic position, use swe_sol_eclipse_when_loc().

```
int32 swe_sol_eclipse_when_loc(
  double tjd_start,   // start date for search, Jul. day UT */
  int32 ifl,      // ephemeris flag */
  double *geopos, // 3 doubles for geographic lon, lat, height.
          // eastern longitude is positive,
          // western longitude is negative,
          // northern latitude is positive,
          // southern latitude is negative */
  double *tret,   // return array, 10 doubles, see below */
  double *attr,   // return array, 20 doubles, see below */
  AS_BOOL backward,   // TRUE, if backward search */
  char *serr      // return error string */
);
```

The function returns:

```
retflag = -1 (ERR) on error (e.g. if swe_calc() for sun or moon fails)
    =  0 if no eclipse
    = SE_ECL_TOTAL or SE_ECL_ANNULAR or SE_ECL_PARTIAL
      combined with bit flags
      SE_ECL_VISIBLE,
      SE_ECL_MAX_VISIBLE,
      SE_ECL_1ST_VISIBLE, SE_ECL_2ND_VISIBLE
      SE_ECL_3ST_VISIBLE, SE_ECL_4ND_VISIBLE

  tret[0] time of maximum eclipse
  tret[1] time of first contact
  tret[2] time of second contact
  tret[3] time of third contact
  tret[4] time of forth contact
  tret[5] time of sunrise between first and forth contact
  tret[6] time of sunset between first and forth contact

  attr[0] fraction of solar diameter covered by moon;
      with total/annular eclipses, it results in magnitude acc. to IMCCE.
  attr[1] ratio of lunar diameter to solar one
  attr[2] fraction of solar disc covered by moon (obscuration)
  attr[3] diameter of core shadow in km
  attr[4] azimuth of sun at tjd
  attr[5] true altitude of sun above horizon at tjd
  attr[6] apparent altitude of sun above horizon at tjd
  attr[7] elongation of moon in degrees
  attr[8] magnitude acc. to NASA;
    = attr[0] for partial and attr[1] for annular and total eclipses
  attr[9] saros series number (if available; otherwise -99999999)
  attr[10] saros series member number (if available; otherwise -99999999)
```

7.4. swe_sol_eclipse_where()

This function can be used to find out the geographic position, where, for a given time, a central eclipse is central or where a non-central eclipse is maximal.

If you want to draw the eclipse path of a total or annular eclipse on a map, first compute the start and end time of the total or annular phase with **swe_sol_eclipse_when_glob()**, then call **swe_sol_eclipse_how()** for several time intervals to get geographic positions on the central path. The northern and southern limits of the umbra and penumbra are not implemented yet.

```
int32 swe_sol_eclipse_where(
    double tjd_ut,  // time, Jul. day UT
    int32 ifl,      // ephemeris flag
    double *geopos,     // array of 10 (!!) doubles
    double *attr,   // return array, 20 doubles, see below
    char *serr      // return error string
);
```

The function returns:

```
retflag =  -1 (ERR) on error (e.g. if swe_calc() for sun or moon fails)
    =  0, if no eclipse is visible at geogr. position.
    =  SE_ECL_TOTAL, or
```

```
        SE_ECL_ANNULAR
        SE_ECL_TOTAL | SE_ECL_CENTRAL
        SE_ECL_TOTAL | SE_ECL_NONCENTRAL
        SE_ECL_ANNULAR | SE_ECL_CENTRAL
        SE_ECL_ANNULAR | SE_ECL_NONCENTRAL
        SE_ECL_PARTIAL

  geopos[0]: geographic longitude of central line
  geopos[1]: geographic latitude of central line
  not implemented so far:
    geopos[2]: geographic longitude of northern limit of umbra
    geopos[3]: geographic latitude of northern limit of umbra
    geopos[4]: geographic longitude of southern limit of umbra
    geopos[5]: geographic latitude of southern limit of umbra
    geopos[6]: geographic longitude of northern limit of penumbra
    geopos[7]: geographic latitude of northern limit of penumbra
    geopos[8]: geographic longitude of southern limit of penumbra
    geopos[9]: geographic latitude of southern limit of penumbra
    eastern longitudes are positive,
    western longitudes are negative,
    northern latitudes are positive,
    southern latitudes are negative

  attr[0] fraction of solar diameter covered by the moon
  attr[1] ratio of lunar diameter to solar one
  attr[2] fraction of solar disc covered by moon (obscuration)
  attr[3] diameter of core shadow in km
  attr[4] azimuth of sun at tjd
  attr[5] true altitude of sun above horizon at tjd
  attr[6] apparent altitude of sun above horizon at tjd
  attr[7] angular distance of moon from sun in degrees
  attr[8] eclipse magnitude
      (= attr[0] or attr[1] depending on eclipse type)
  attr[9] saros series number (if available; otherwise -99999999)
  attr[10] saros series member number (if available; otherwise -99999999)
```

declare as attr[20] and geopos[10] !

7.5. swe_sol_eclipse_how()

To calculate the attributes of an eclipse for a given geographic position and time:

```
int32 swe_sol_eclipse_how(
  double tjd_ut,  // time, Jul. day UT
  int32 ifl,      // ephemeris flag
  double *geopos,     // geogr. longitude, latitude, height above sea, array of 3 doubles
          // eastern longitude is positive,
          // western longitude is negative,
          // northern latitude is positive,
          // southern latitude is negative
  double *attr,   // return array, 20 doubles, see below
  char *serr      // return error string
);
```

The function returns:

```
retflag =  -1 (ERR) on error (e.g. if swe_calc() for sun or moon fails)
    =  SE_ECL_TOTAL or SE_ECL_ANNULAR or SE_ECL_PARTIAL
    =  0, if no eclipse is visible at geogr. position.

  attr[0] fraction of solar diameter covered by moon;
  with total/annular eclipses, it results in magnitude acc. to IMCCE.
  attr[1] ratio of lunar diameter to solar one
  attr[2] fraction of solar disc covered by moon (obscuration)
  attr[3] diameter of core shadow in km
  attr[4] azimuth of sun at tjd
  attr[5] true altitude of sun above horizon at tjd
  attr[6] apparent altitude of sun above horizon at tjd
  attr[7] elongation of moon in degrees
  attr[8] magnitude acc. to NASA;
      = attr[0] for partial and attr[1] for annular and total eclipses
  attr[9] saros series number (if available; otherwise -99999999)
  attr[10] saros series member number (if available; otherwise -99999999)
```

7.6. swe_lun_occult_when_glob()

To find the next occultation of a planet or star by the moon globally (not for a particular geographic location):

The same function can also be used for global solar eclipses instead of **swe_sol_eclipse_when_glob()**, with ipl = SE_SUN, but is a bit less efficient.

```
int32 swe_lun_occult_when_glob(
    double tjd_start, // start date for search, Jul. day UT
    int32 ipl,    // planet number of occulted body (ignored if starname given)
    char  *starname,  // if NULL or empty, occultation of ipl is searched
    int32 ifl,    // ephemeris flag
    int32 ifltype,    // eclipse type wanted: SE_ECL_TOTAL etc. or 0, if any > eclipse type
    double *tret,     // return array, 10 doubles, see below
    AS_BOOL backward, // 0 = forward, 1 = backward search
              // can be combined with SE_ECL_ONE_TRY to try only one full Moon orbit
    char *serr    // return error string
);
```

If you want to have only one conjunction of the moon with the body tested, add the following flag: backward |= SE_ECL_ONE_TRY. If this flag is not set, the function will search for an occultation until it finds one. For bodies with ecliptical latitudes > 5, the function may search successlessly until it reaches the end of the ephemeris.

The function returns:

```
retflag = -1 (ERR) on error (e.g. if **swe_calc()** for sun or moon fails)
    = 0 (if no occultation / eclipse has been found)
    = SE_ECL_TOTAL or SE_ECL_ANNULAR or SE_ECL_PARTIAL or SE_ECL_ANNULAR_TOTAL
      combined with
      SE_ECL_CENTRAL
      SE_ECL_NONCENTRAL

  tret[0] time of maximum eclipse
  tret[1] time, when eclipse takes place at local apparent noon
  tret[2] time of eclipse begin
```

```
tret[3] time of eclipse end
tret[4] time of totality begin
tret[5] time of totality end
tret[6] time of center line begin
tret[7] time of center line end
tret[8] time when annular-total eclipse becomes total, not implemented so far
tret[9] time when annular-total eclipse becomes annular, again not implemented so far
```

declare as tret[10] at least!

7.7. swe_lun_occult_when_loc()

To find the next occultation of a planet or star by the moon for a given location.

The same function can also be used for local solar eclipses instead of swe_sol_eclipse_when_loc(), with ipl = SE_SUN, but is a bit less efficient.

```
int32 swe_lun_occult_when_loc(
  double tjd_start,   // start date for search, Jul. day UT */
  int32 ipl,      // planet number of occulted body (ignored if starname given)
  char  *starname,    // if NULL or empty, occultation of ipl is searched
  int32 ifl,      // ephemeris flag */
  double *geopos, // 3 doubles for geographic lon, lat, height.
          // eastern longitude is positive,
          // western longitude is negative,
          // northern latitude is positive,
          // southern latitude is negative */
  double *tret,   // return array, 10 doubles, see below */
  double *attr,   // return array, 20 doubles, see below */
  AS_BOOL backward,   // TRUE, if backward search */
  char *serr      // return error string */
);
```

Occultations of some stars may be very rare or do not occur at all. Usually the function searches an event until it finds one or reaches the end of the ephemeris. In order to avoid endless loops, the function can be called using the flag ifl |= SE_ECL_ONE_TRY. If called with this flag, the function searches the next date when the Moon is in conjunction with the object and finds out whether it is an occultation. The function does not check any other conjunctions in the future or past.

- If the return value is > 0, there is an occultation and tret and attr contain the information about it;
- If the return value is = 0, there is no occupation; tret[0] contains the date of closest conjunction;
- If the return value is = -1, there is an error.

In order to find events in a particular time range (tjd_start < tjd < tjd_stop), one can write a loop and call the function as often as date (tjd < tjd_stop). After each call, increase the tjd = tret[0] + 2.

If one has a set of stars or planets for which one wants to find occultations for the same time range, one has to run the same loop for each of these object. If the events have to be listed in chronological order, one has to sort them before output.

The function returns:

```
retflag = -1 (ERR) on error (e.g. if **swe_calc()** for sun or moon fails)
    = 0 (if no occultation/no eclipse found)
    = SE_ECL_TOTAL or SE_ECL_PARTIAL or SE:ECL_ANNULAR (only for ipl = SE_SUN)
      combined with
      SE_ECL_VISIBLE,
```

```
      SE_ECL_MAX_VISIBLE,
      SE_ECL_1ST_VISIBLE, SE_ECL_2ND_VISIBLE
      SE_ECL_3ST_VISIBLE, SE_ECL_4ND_VISIBLE

    tret[0] time of maximum eclipse
    tret[1] time of first contact
    tret[2] time of second contact
    tret[3] time of third contact
    tret[4] time of forth contact
    tret[5] time of sunrise between first and forth contact (not implemented so far)
    tret[6] time of sunset between first and forth contact (not implemented so far)

    attr[0] fraction of solar diameter covered by moon (magnitude)
    attr[1] ratio of lunar diameter to solar one
    attr[2] fraction of solar disc covered by moon (obscuration)
    attr[3] diameter of core shadow in km
    attr[4] azimuth of sun at tjd
    attr[5] true altitude of sun above horizon at tjd
    attr[6] apparent altitude of sun above horizon at tjd
    attr[7] elongation of moon in degrees
```

7.8. swe_lun_occult_where ()

Similar to **swe_sol_eclipse_where()**, this function can be used to find out the geographic position, where, for a given time, a central eclipse is central or where a non-central eclipse is maximal. With occultations, it tells us, at which geographic location the occulted body is in the middle of the lunar disc or closest to it. Because occultations are always visible from a very large area, this is not very interesting information. But it may become more interesting as soon as the limits of the umbra (and penumbra) will be implemented.

```
int32 swe_lun_occult_where(
    double tjd_ut,  // time, Jul. day UT
    int32 ipl,      // planet number of occulted body (ignored if starname given)
    char  *starname,    // if NULL or empty, occultation of ipl is searched
    int32 ifl,      // ephemeris flag
    double *geopos,     // array of 10 (!!) doubles
    double *attr,   // return array, 20 doubles, see below
    char *serr      // return error string
);
```

The function returns:

```
  retflag = -1 (ERR) on error (e.g. if swe_calc() for sun or moon fails)
      = 0 if there is no solar eclipse (occultation) at tjd
      = SE_ECL_TOTAL or
        SE_ECL_ANNULAR
        SE_ECL_TOTAL | SE_ECL_CENTRAL
        SE_ECL_TOTAL | SE_ECL_NONCENTRAL
        SE_ECL_ANNULAR | SE_ECL_CENTRAL
        SE_ECL_ANNULAR | SE_ECL_NONCENTRAL
        SE_ECL_PARTIAL

  geopos[0]: geographic longitude of central line
  geopos[1]: geographic latitude of central line
    not implemented so far:
```

```
  geopos[2]: geographic longitude of northern limit of umbra
  geopos[3]: geographic latitude of northern limit of umbra
  geopos[4]: geographic longitude of southern limit of umbra
  geopos[5]: geographic latitude of southern limit of umbra
  geopos[6]: geographic longitude of northern limit of penumbra
  geopos[7]: geographic latitude of northern limit of penumbra
  geopos[8]: geographic longitude of southern limit of penumbra
  geopos[9]: geographic latitude of southern limit of penumbra

attr[0] fraction of object's diameter covered by moon (magnitude)
attr[1] ratio of lunar diameter to object's diameter
attr[2] fraction of object's disc covered by moon (obscuration)
attr[3] diameter of core shadow in km
attr[4] azimuth of object at tjd
attr[5] true altitude of object above horizon at tjd
attr[6] apparent altitude of object above horizon at tjd
attr[7] angular distance of moon from object in degrees
```

declare as attr[20] and geopos[10] !

7.9. swe_lun_eclipse_when_loc()

To find the next lunar eclipse observable from a given geographic position:

```
int32 swe_lun_eclipse_when_loc(
  double tjd_start,   // start date for search, Jul. day UT */
  int32 ifl,      // ephemeris flag */
  double *geopos, // 3 doubles for geographic lon, lat, height.
          // eastern longitude is positive,
          // western longitude is negative,
          // northern latitude is positive,
          // southern latitude is negative */
  double *tret,   // return array, 10 doubles, see below */
  double *attr,   // return array, 20 doubles, see below */
  AS_BOOL backward,   // TRUE, if backward search */
  char *serr      // return error string */
);
```

If your code does not work, please study the sample code in swetest.c.

The function returns:

```
retflag = -1 (ERR) on error (e.g. if swe_calc() for sun or moon fails)
    = 0 if there is no lunar eclipse at tjd
    = SE_ECL_TOTAL or SE_ECL_PENUMBRAL or SE_ECL_PARTIAL

tret[0] time of maximum eclipse
tret[1]
tret[2] time of partial phase begin (indices consistent with solar eclipses)
tret[3] time of partial phase end
tret[4] time of totality begin
tret[5] time of totality end
tret[6] time of penumbral phase begin
tret[7] time of penumbral phase end
tret[8] time of moonrise, if it occurs during the eclipse
tret[9] time of moonset, if it occurs during the eclipse
```

```
attr[0] umbral magnitude at tjd
attr[1] penumbral magnitude
attr[4] azimuth of moon at tjd
attr[5] true altitude of moon above horizon at tjd
attr[6] apparent altitude of moon above horizon at tjd
attr[7] distance of moon from opposition in degrees
attr[8] umbral magnitude at tjd (= attr[0])
attr[9] saros series number (if available; otherwise -99999999)
attr[10] saros series member number (if available; otherwise -99999999) */
```

7.10. swe_lun_eclipse_when()

To find the next lunar eclipse:

```
int32 swe_lun_eclipse_when(
    double tjd_start, // start date for search, Jul. day UT
    int32 ifl,    // ephemeris flag
    int32 ifltype,    // eclipse type wanted: SE_ECL_TOTAL etc. or 0, if any > eclipse type
    double *tret,     // return array, 10 doubles, see below
    AS_BOOL backward, // TRUE, if backward search
    char *serr    // return error string
);
```

Recommended values for ifltype:

```
// search for any lunar eclipse, no matter which type
  ifltype = 0;
// search a total lunar eclipse
  ifltype = SE_ECL_TOTAL;
//  search a partial lunar eclipse
  ifltype = SE_ECL_PARTIAL;
// search a penumbral lunar eclipse
  ifltype = SE_ECL_PENUMBRAL;
```

The function returns:

```
retflag = -1 (ERR) on error (e.g. if swe_calc() for sun or moon fails)
    = SE_ECL_TOTAL or SE_ECL_PENUMBRAL or SE_ECL_PARTIAL

tret[0] time of maximum eclipse
tret[1]
tret[2] time of partial phase begin (indices consistent with solar eclipses)
tret[3] time of partial phase end
tret[4] time of totality begin
tret[5] time of totality end
tret[6] time of penumbral phase begin
tret[7] time of penumbral phase end
```

7.11. swe_lun_eclipse_how()

This function computes the attributes of a lunar eclipse at a given time:

```
int32 swe_lun_eclipse_how(
  double tjd_ut,  // time, Jul. day UT
```

```
  int32 ifl,      // ephemeris flag
  double *geopos,     // geogr. longitude, latitude, height above sea, array of 3 doubles
          // eastern longitude is positive,
          // western longitude is negative,
          // northern latitude is positive,
          // southern latitude is negative
  double *attr,   // return array, 20 doubles, see below
  char *serr      // return error string
);
```

The function returns:

```
retflag = -1 (ERR) on error (e.g. if swe_calc() for sun or moon fails)
    = 0 if there is no eclipse
    = SE_ECL_TOTAL or SE_ECL_PENUMBRAL or SE_ECL_PARTIAL

attr[0] umbral magnitude at tjd
attr[1] penumbral magnitude
attr[4] azimuth of moon at tjd. Not implemented so far
attr[5] true altitude of moon above horizon at tjd. Not implemented so far
attr[6] apparent altitude of moon above horizon at tjd. Not implemented so far
attr[7] distance of moon from opposition in degrees
attr[8] eclipse magnitude (= attr[0])
attr[9] saros series number (if available; otherwise -99999999)
attr[10] saros series member number (if available; otherwise -99999999)
```

declare as attr[20] at least!

8. Date and time conversion functions

8.1. Calendar date and Julian day: swe_julday(), swe_date_conversion(), swe_revjul()

These functions are needed to convert calendar dates to the astronomical time scale which measures time in Julian days.

swe_julday() and **swe_date_conversion()** compute a Julian day number from year, month, day, and hour.

swe_date_conversion() checks in addition whether the date is legal. It returns OK or ERR.

swe_revjul() is the reverse function of **swe_julday()**. It computes year, month, day and hour from a Julian day number.

Julian day number from year, month, day, hour

```
double swe_julday(
  int year,
  int month,
  int day,
  double hour,
  int gregflag // Gregorian calendar: 1 = SE_GREG_CAL, Julian calendar: 0 = SE_JUL_CAL
);
```

The variable gregflag tells the function whether the input date is Julian calendar (gregflag = SE_JUL_CAL) or Gregorian calendar (gregflag = SE_GREG_CAL).

Usually, you will set gregflag = SE_GREG_CAL.

```
int swe_date_conversion(
  int year,
  int month,
  int day,
  double hour,  // hours (decimal, with fraction)
  char c,   // calendar 'g'[regorian] or 'j'[ulian]
  double *tjd   // target address for Julian day
);
```

Please note that this second function does not expect a boolean greg_flag, but a character 'g' or j'. Anything other then 'j' will be considered like 'g'.

Year, month, day, hour from Julian day number

```
void swe_revjul(
  double tjd,   // Julian day number
  int gregflag, // Gregorian calendar: 1, Julian calendar: 0
  int *year,    // target addresses for year, etc.
  int *month,
  int *day,
  double *hour
);
```

The Julian day number has nothing to do with Julius Cesar, who introduced the Julian calendar, but was invented by the monk Julianus. The Julian day number tells for a given date the number of days that have passed since the creation of the world which was then considered to have happened on 1 Jan - 4712 at noon. E.g. the 1.1.1900 corresponds to the Julian day number 2415020.5.

Midnight has always a JD with fraction 0.5, because traditionally the astronomical day started at noon. This was practical because then there was no change of date during a night at the telescope. From this comes also the fact that noon ephemerides were printed before midnight ephemerides were introduced early in the 20th century.

8.2. UTC and Julian day: swe_utc_time_zone(), swe_utc_to_jd(), swe_jdet_to_utc(), swe_jdut1_to_utc()

The following functions, which were introduced with Swiss Ephemeris version 1.76, do a similar job as the functions described under 8.1. The difference is that input and output times are Coordinated Universal Time (UTC). For transformations between wall clock (or arm wrist) time and Julian Day numbers, these functions are more correct. The difference is always less than 1 second, though.

Use these functions to convert:

- local time to UTC and UTC to local time;
- UTC to a Julian day number, and
- a Julian day number to UTC.

Past leap seconds are hard coded in the Swiss Ephemeris. Future leap seconds can be specified in the file seleapsec.txt, see ch. 8.3.

NOTE: in case of leap seconds, the input or output time may be 60.9999 seconds. in UTC, there exist from timt to time minutes which have only 59 seconds, or minutes which have 61 seconds. Input or output forms have to allow for this.

swe_utc_time_zone() Local time to UTC and UTC to local time

```
/* transform local time to UTC or UTC to local time
* input:
* iyear ... dsec date and time
* d_timezone timezone offset
* output:
* iyear_out ... dsec_out
*
* For time zones east of Greenwich, d_timezone is positive.
* For time zones west of Greenwich, d_timezone is negative.
*
* For conversion from local time to utc, use +d_timezone.
```

```
* For conversion from utc to local time, use -d_timezone.
*/

void swe_utc_timezone(
  int32 iyear,
  int32 imonth,
  int32 iday,
  int32 ihour,
  int32 imin,
  double dsec,
  double d_timezone,
  int32 *iyear_out,
  int32 *imonth_out,
  int32 *iday_out,
  int32 *ihour_out,
  int32 *imin_out,
  double *dsec_out
);
```

Please note that the caller must know the local timezone offset. There exists no Swiss Ephemeris function which provide time zone information for a given date and location.

swe_utc_to_jd() UTC to jd (TT and UT1)

```
/* input: date and time (wall clock time), calendar flag.
* output: an array of doubles with Julian Day number in ET (TT) and UT (UT1)
* an error message (on error)
* The function returns OK or ERR.
*/

void swe_utc_to_jd(
  int32 iyear,
  int32 imonth,
  int32 iday,
  int32 ihour,
  int32 imin,
  double dsec,  // NOTE: second is a decimal
  gregflag,     // Gregorian calendar: 1, Julian calendar: 0
  dret,     // return array, two doubles:
        // dret[0] = Julian day in ET (TT)
        // dret[1] = Julian day in UT (UT1)
  serr      //  error string
);
```

swe_jdet_to_utc() TT (ET1) to UTC

```
/* input: Julian day number in ET (TT), calendar flag
* output: year, month, day, hour, min, sec in UTC */

void swe_jdet_to_utc(
  double tjd_et,// Julian day number in ET (TT)
  gregflag,     // Gregorian calendar: 1, Julian calendar: 0
  int32 *iyear,
```

```
  int32 *imonth,
  int32 *iday,
  int32 *ihour,
  int32 *imin,
  double *dsec  // NOTE: second is a decimal
);
```

swe_jdut1_to_utc() UT (UT1) to UTC

```
/* input: Julian day number in UT (UT1), calendar flag
* output: year, month, day, hour, min, sec in UTC */

void swe_jdut1_to_utc(
  double tjd_ut, // Julian day number in UT (UTC)
  gregflag,      // Gregorian calendar: 1, Julian calendar: 0
  int32 *iyear,
  int32 *imonth,
  int32 *iday,
  int32 *ihour,
  int32 *imin,
  double *dsec  // NOTE: second is a decimal
);
```

Example: How do I get correct planetary positions, sidereal time, and house cusps, starting from a wall clock date and time?

```
int32 iday, imonth, iyear, ihour, imin, retval;
int32 iday_utc, imonth_utc, iyear_utc, ihour_utc, imin_utc, retval;
int32 gregflag ] =  SE_GREG_CAL;
double d_timezone = 5.5; // time zone = Indian Standard Time; east is positive
double dsec, dsec_utc, tjd_et, tjd_ut;
double dret[2];
char serr[256];
...

// if date and time is in time zone different from UTC,
// the time zone offset must be subtracted first in order to get UTC:
swe_utc_time_zone(iyear, imonth, iday, ihour, imin, dsec,
d_timezone, &iyear_utc, &imonth_utc, &iday_utc, &ihour_utc, &imin_utc, &dsec_utc);

// calculate Julian day number in UT (UT1) and ET (TT) from UTC
retval = swe_utc_to_jd(iyear_utc, imonth_utc, iday_utc, ihour_utc,
imin_utc, dsec_utc, gregflag, dret, serr);

if (retval == ERR) {
  fprintf(stderr, serr); / error handling
  retur ERR;
}
tjd_et = dret[0]; /* this is ET (TT) */
tjd_ut = dret[1]; /* this is UT (UT1) */

// calculate planet with tjd_et
swe_calc(tjd_et, ...);
```

```
// calculate houses with tjd_ut
swe_houses(tjd_ut, ...)
```

Example: How do you get the date and wall clock time from a Julian day number?

```
// Depending on whether you have tjd_et (Julian day as ET (TT)) or tjd_ut
// (Julian day as UT (UT1)), use one of the two functions
// swe_jdet_to_utc() or swe_jdut1_to_utc().

...

// first, we calculate UTC from TT (ET)
swe_jdet_to_utc(tjd_et, gregflag, &iyear_utc, &imonth_utc,
&iday_utc, &ihour_utc, &imin_utc, &dsec_utc);

// now, UTC to local time (note the negative sign before d_timezone):
swe_utc_time_zone(iyear_utc, imonth_utc, iday_utc, ihour_utc, imin_utc, dsec_utc,
-d_timezone, &iyear, &imonth, &iday, &ihour, &imin, &dsec);
```

8.3. Handling of leap seconds and the file seleapsec.txt

The insertion of leap seconds is not known in advance. We will update the Swiss Ephemeris whenever the IERS announces that a leap second will be inserted. However, if the user does not want to wait for our update or does not want to download a new version of the Swiss Ephemeris, he can create a file **seleapsec.txt** in the ephemeris directory. The file looks as follows (lines with # are only comments):

```
# This file contains the dates of leap seconds to be taken into account
# by the Swiss Ephemeris.
# For each new leap second add the date of its insertion in the format
# yyyymmdd, e.g. "20081231" for 31 december 2008.
# The leap second is inserted at the end of the day.
20081231
```

Note: there is currently no provision to handle negative leap seconds via this file. No leap seconds have been inserted since 2016, and it is possible that negative leap seconds will appear in 2022 or later.

Before 1972, **swe_utc_to_jd()** treats its input time as UT1.

NOTE: UTC was introduced in 1961. From 1961 - 1971, the length of the UTC second was regularly changed, so that UTC remained very close to UT1.

From 1972 on, input time is treated as UTC.

If delta_t - nleap - 32.184 > 1, the input time is treated as UT1.

NOTE: Like this we avoid errors greater than 1 second in case that the leap seconds table (or the Swiss Ephemeris version) is not updated for a long time.

8.4. Mean solar time versus True solar time: swe_time_equ(), swe_lmt_to_lat(), swe_lat_to_lmt()

Universal Time (UT or UTC) is based on Mean Solar Time, AKA Local Mean Time, which is a uniform measure of time. A day has always the same length, independent of the time of the year.

In the centuries before mechanical clocks where used, when the reckoning of time was mostly based on sun dials, the True Solar Time was used, also called Local Apparent Time.

The difference between Local Mean Time and Local Apparent Time is called the *equation of time*. This difference can become as large as 20 minutes.

If a historical date was noted in Local Apparent Time, it must first be converted to Local Mean Time by applying the equation of time, before it can be used to compute Universal Time (for the houses) and finally [Ephemeris Time] (for the planets).

This conversion can be done using the function **swe_lat_to_lmt()**. The reverse function is **swe_lmt_to_lat()**. If required, the equation of time itself, i. e. the value e = LAT - LMT, can be calculated using the function **swe_time_equ()**:

Equation of time: swe_time_equ()

```
/* function returns the difference between local apparent and local mean time.
e = LAT -- LMT. tjd_et is ephemeris time */

int swe_time_equ(
  double tjd_et,
  double *e,
  char *serr
);

returns OK or ERR
```

convert Local Apparent Time (LAT) to Local Mean Time (LMT): swe_lat_to_lmt()

```
// tjd_lmt and tjd_lat are a Julian day number
// geolon is geographic longitude, where eastern
// longitudes are positive, western ones negative

int32 swe_lat_to_lmt(
  double tjd_lat,
  double geolon,
  double *tjd_lmt,
  char *serr
);

returns OK or ERR
```

convert Local Mean Time (LMT) to Local Apparent Time (LAT): swe_lmt_to_lat()

```
int32 swe_lmt_to_lat(
  double tjd_lmt,
  double geolon,
  double *tjd_lat,
  char *serr
);

returns OK or ERR
```

9. Delta T-related functions

```
// delta t from Julian day number

double swe_deltat_ex(
  double tjd,
  int32 ephe_flag,
  char *serr
);

// delta t from Julian day number
double swe_deltat(
  double tjd
);

// get tidal acceleration used in swe_deltat()
double swe_get_tid_acc(void);

// set tidal acceleration to be used in swe_deltat()
void swe_set_tid_acc(
  double t_acc
);

// set fixed Delta T value to be returned by swe_deltat()
void swe_set_delta_t_userdef(
  double t_acc
);
```

The Julian day number, you compute from a birth date, will be Universal Time (UT, former GMT) and can be used to compute the star time and the houses. However, for the planets and the other factors, you have to convert UT to Ephemeris time (ET):

9.1. swe_deltat_ex()

tjde = tjd + **swe_deltat_ex**(tjd, ephe_flag, serr);

where

tjd = Julian day in UT, tjde = in ET

ephe_flag = ephemeris flag (one of SEFLG_SWIEPH, SEFLG_JPLEPH, SEFLG_MOSEPH)

serr = string pointer for warning messages.

If the function is called with SEFLG_SWIEPH before calling **swe_set_ephe_path()**, or with or SEFLG_JPLEPH before calling **swe_set_jpl_file()**, then the function returns a warning.

The calculation of ephemerides in UT depends on Delta T, which depends on the ephemeris-inherent value of the tidal acceleration of the Moon. The function **swe_deltat_ex()** can provide ephemeris-dependent values of Delta T and is therefore better than the old function **swe_deltat()**, which has to make an uncertain guess of what ephemeris is being used. One warning must be made, though:

It is **not recommended** to use a mix of old and new ephemeris files sepl*.se1, semo*.se1, seas*.se1, because the old files were based on JPL Ephemeris DE406, whereas the new ones are based on DE431, and both ephemerides have a different inherent tidal acceleration of the Moon. A mixture of old and new ephemeris files may lead to inconsistent ephemeris output. Using old asteroid files se99999.se1 together with new ones, can be tolerated, though.

9.2. swe_deltat()

tjde = tjd + **swe_deltat**(tjd);

where

tjd = Julian day in UT, tjde = in ET

This function is safe only:

- if your software consistently uses the same ephemeris flag;
- if your software consistently uses the same ephemeris files (with SEFLG_SWIEPH and SEFLG_MOSEPH);
- if you first call **swe_set_ephe_path()** (with SEFLG_SWIEPH) and **swe_set_jpl_file()** (with SEFLG_JPLEPH).

(Also, it is safe if you first call **swe_set_tid_acc()** with the tidal acceleration you want. However, please do not use this function unless you really know what you are doing.)

For best control of the values returned, use function **swe_deltat_ex()** instead (see 9.1 above).

The calculation of ephemerides in UT depends on Delta T, which depends on the ephemeris-inherent value of the tidal acceleration of the Moon. In default mode, the function **swe_deltat()** automatically tries to find the required values. Two warnings must be made, though:

1. It is **not recommended** to use a mix of old and new ephemeris files sepl*.se1, semo*.se1, seas*.se1, because the old files were based on JPL Ephemeris DE406, whereas the new ones are based on DE431, and both ephemerides have a different inherent tidal acceleration of the Moon. A mixture of old and new ephemeris files may lead to inconsistent ephemeris output. Using old asteroid files se99999.se1 together with new ones, can be tolerated, though.
2. The function **swe_deltat()** uses a default value of tidal acceleration (that of DE431). However, after calling some older ephemeris, like Moshier ephemeris, DE200, or DE406, **swe_deltat() might provide** slightly different values.

In case of troubles related to these two points, it is recommended to:

- either use the function **swe_deltat_ex()**;
- or control the value of the tidal acceleration using the functions **swe_set_tid_acc()** and **swe_get_tid_acc()**.

9.3. swe_set_tid_acc(), swe_get_tid_acc()

With Swiss Ephemeris versions until 1.80, this function had **always** to be used, if a nonstandard ephemeris like DE200 or DE421 was used.

Since Swiss Ephemeris version 2.00, this function is usually not needed, because the value is automatically set according to the ephemeris files selected or available. However, under certain circumstances that are described in the section "9.2 swe_deltat()", the user may want to control the tidal acceleration himself.

To find out the value of the tidal acceleration currently used, call the function

acceleration = **swe_get_tid_acc**();

In order to set a different value, use the function

swe_set_tid_acc(acceleration);

The values that acceleration can have are listed in swephexp.h. (e.g. SE_TIDAL_200, etc.)

Once the function **swe_set_tid_acc()** has been used, the automatic setting of tidal acceleration is blocked. In order to unblock it again, call

swe_set_tid_acc(SE_TIDAL_AUTOMATIC);

9.4. swe_set_delta_t_userdef()

This function allows the user to set a fixed Delta T value that will be returned by **swe_deltat()** or **swe_deltat_ex()**.

The same Delta T value will then be used by **swe_calc_ut()**, eclipse functions, heliacal functions, and all functions that require UT as input time.

In order to return to automatic Delta T, call this function with the following value:

swe_set_delta_t_userdef(SE_DELTAT_AUTOMATIC);

9.5. Future updates of Delta T and the file swe_deltat.txt

Delta T values for future years can only be estimated. Strictly speaking, the Swiss Ephemeris has to be updated every year after the new Delta T value for the past year has been published by the IERS. We will do our best and hope to update the Swiss Ephemeris every year. However, if the user does not want to wait for our update or does not download a new version of the Swiss Ephemeris he can add new Delta T values in the file **swe_deltat.txt**, which has to be located in the Swiss Ephemeris ephemeris path.

```
# This file allows make new Delta T known to the Swiss Ephemeris.
# Note, these values override the values given in the internal Delta T
# table of the Swiss Ephemeris.
# Format: year and seconds (decimal)

2003 64.47
2004 65.80
2005 66.00
2006 67.00
2007 68.00
2008 68.00
2009 69.00
```

10. The function swe_set_topo() for topocentric planet positions

```
void swe_set_topo(
  // 3 doubles for geogr. longitude, latitude, height above sea.
  double geolon, // eastern longitude is positive, western longitude is negative
  double geolat, // northern latitude is positive, southern latitude is negative
  double altitude
);
```

This function must be called before topocentric planet positions for a certain birth place can be computed. It tells Swiss Ephemeris, what geographic position is to be used. Geographic longitude geolon and latitude geolat must be in degrees, the altitude above sea must be in meters. Neglecting the altitude can result in an error of about 2 arc seconds with the Moon and at an altitude 3000 m. After calling **swe_set_topo()**, add SEFLG_TOPOCTR to iflag and call **swe_calc()** as with an ordinary computation. E.g.:

```
swe_set_topo(geo_lon, geo_lat, altitude_above_sea);
iflag |= SEFLG_TOPOCTR;
for (i = 0; i < NPLANETS; i++) {
  iflgret = swe_calc(tjd, ipl, iflag, xp, serr);
  printf("%f\n", xp[0]);
}
```

The parameters set by **swe_set_topo()** survive **swe_close().**

12. Sidereal mode functions

12.1. swe_set_sid_mode()

```
void swe_set_sid_mode(
  int32 sid_mode,
  double t0,
  double ayan_t0
);
```

This function can be used to specify the mode for sidereal computations.

swe_calc() or **swe_fixstar()** has then to be called with the bit SEFLG_SIDEREAL.

If **swe_set_sid_mode()** is not called, the default ayanamsha (Fagan/Bradley) is used.

If a predefined mode is wanted, the variable sid_mode has to be set, while t0 and ayan_t0 are not considered, i.e. can be 0. The predefined sidereal modes are:

```
#define SE_SIDM_FAGAN_BRADLEY 0
#define SE_SIDM_LAHIRI 1
#define SE_SIDM_DELUCE 2
#define SE_SIDM_RAMAN 3
#define SE_SIDM_USHASHASHI 4
#define SE_SIDM_KRISHNAMURTI 5
#define SE_SIDM_DJWHAL_KHUL 6
#define SE_SIDM_YUKTESHWAR 7
#define SE_SIDM_JN_BHASIN 8
#define SE_SIDM_BABYL_KUGLER1 9
#define SE_SIDM_BABYL_KUGLER2 10
#define SE_SIDM_BABYL_KUGLER3 11
#define SE_SIDM_BABYL_HUBER 12
#define SE_SIDM_BABYL_ETPSC 13
#define SE_SIDM_ALDEBARAN_15TAU 14
#define SE_SIDM_HIPPARCHOS 15
#define SE_SIDM_SASSANIAN 16
#define SE_SIDM_GALCENT_0SAG 17
#define SE_SIDM_J2000 18
#define SE_SIDM_J1900 19
#define SE_SIDM_B1950 20
#define SE_SIDM_SURYASIDDHANTA 21
#define SE_SIDM_SURYASIDDHANTA_MSUN 22
#define SE_SIDM_ARYABHATA 23
#define SE_SIDM_ARYABHATA_MSUN 24
#define SE_SIDM_SS_REVATI 25
#define SE_SIDM_SS_CITRA 26
```

```
#define SE_SIDM_TRUE_CITRA 27
#define SE_SIDM_TRUE_REVATI 28
#define SE_SIDM_TRUE_PUSHYA 29
#define SE_SIDM_GALCENT_RGBRAND 30
#define SE_SIDM_GALEQU_IAU1958 31
#define SE_SIDM_GALEQU_TRUE 32
#define SE_SIDM_GALEQU_MULA 33
#define SE_SIDM_GALALIGN_MARDYKS 34
#define SE_SIDM_TRUE_MULA 35
#define SE_SIDM_GALCENT_MULA_WILHELM 36
#define SE_SIDM_ARYABHATA_522 37
#define SE_SIDM_BABYL_BRITTON 38
#define SE_SIDM_TRUE_SHEORAN 39
#define SE_SIDM_GALCENT_COCHRANE 40
#define SE_SIDM_GALEQU_FIORENZA 41
#define SE_SIDM_VALENS_MOON 42
#define SE_SIDM_LAHIRI_1940 43
#define SE_SIDM_LAHIRI_VP285 44
#define SE_SIDM_KRISHNAMURTI_VP291 45
#define SE_SIDM_LAHIRI_ICRC 46
#define SE_SIDM_USER 255
```

The function **swe_get_ayanamsa_name()** returns the name of the ayanamsha.

```
const char *swe_get_ayanamsa_name(
  int32 isidmode
)
```

namely:

name	number	#define
“Fagan/Bradley”	0	SESIDM_FAGAN_BRADLEY
“Lahiri”	1	SESIDM_LAHIRI
“De Luce”	2	SESIDM_DELUCE
“Raman”	3	SESIDM_RAMAN
“Usha/Shashi”	4	SESIDM_USHASHASHI
“Krishnamurti”	5	SESIDM_KRISHNAMURTI
“Djwhal Khul”	6	SESIDM_DJWHAL_KHUL
“Yukteshwar”	7	SESIDM_YUKTESHWAR
“J.N. Bhasin”	8	SESIDM_JN_BHASIN
“Babylonian/Kugler 1”	9	SESIDM_BABYL_KUGLER1
“Babylonian/Kugler 2”	10	SESIDM_BABYL_KUGLER2
“Babylonian/Kugler 3”	11	SESIDM_BABYL_KUGLER3
“Babylonian/Huber”	12	SESIDM_BABYL_HUBER
“Babylonian/Eta Piscium”	13	SESIDM_BABYL_ETPSC
“Babylonian/Aldebaran = 15 Tau”	14	SESIDM_ALDEBARAN_15TAU
“Hipparchos”	15	SESIDM_HIPPARCHOS
“Sassanian”	16	SESIDM_SASSANIAN
“Galact. Center = 0 Sag”	17	SESIDM_GALCENT_0SAG
“J2000”	18	SESIDM_J2000
“J1900”	19	SESIDM_J1900
“B1950”	20	SESIDM_B1950
“Suryasiddhanta”	21	SESIDM_SURYASIDDHANTA
“Suryasiddhanta, mean Sun”	22	SESIDM_SURYASIDDHANTA_MSUN
“Aryabhata”	23	SESIDM_ARYABHATA
“Aryabhata, mean Sun”	24	SESIDM_ARYABHATA_MSUN

name	number	#define
“SS Revati”	25	SESIDM_SS_REVATI
“SS Citra”	26	SESIDM_SS_CITRA
“True Citra”	27	SESIDM_TRUE_CITRA
“True Revati”	28	SESIDM_TRUE_REVATI
“True Pushya (PVRN Rao)”	29	SESIDM_TRUE_PUSHYA
“Galactic Center (Gil Brand)”	30	SESIDM_GALCENT_RGBRAND
“Galactic Equator (IAU1958)”	31	SESIDM_GALEQU_IAU1958
“Galactic Equator”	32	SESIDM_GALEQU_TRUE
“Galactic Equator mid-Mula”	33	SESIDM_GALEQU_MULA
“Skydram (Mardyks)”	34	SESIDM_GALALIGN_MARDYKS
“True Mula (Chandra Hari)”	35	SESIDM_TRUE_MULA
“Dhruva/Gal.Center/Mula (Wilhelm)”	36	SESIDM_GALCENT_MULA_WILHELM
“Aryabhata 522”	37	SESIDM_ARYABHATA_522
“Babylonian/Britton”	38	SESIDM_BABYL_BRITTON
“Vedic/Sheoran”	39	SESIDM_TRUE_SHEORAN
“Cochrane (Gal.Center = 0 Cap)”	40	SESIDM_GALCENT_COCHRANE
“Galactic Equator (Fiorenza)”	41	SESIDM_GALEQU_FIORENZA
“Vettius Valens”	42	SESIDM_VALENS_MOON
“Lahiri 1940”	43	SESIDM_LAHIRI_1940
“Lahiri VP285”	44	SESIDM_LAHIRI_VP285
“Krishnamurti-Senthilathiban”	45	SESIDM_KRISHNAMURTI_VP291
“Lahiri ICRC”	46	SESIDM_LAHIRI_ICRC

For information about the sidereal modes, please read the chapter on sidereal calculations in swisseph.doc.

To define your own sidereal mode, use SE_SIDM_USER (=255) and set the reference date (t0) and the initial value of the ayanamsha (ayan_t0).

```
ayan_t0 = tropical_position_t0 - sidereal_position_t0;
```

Without additional specifications, the traditional method is used. The ayanamsha measured on the ecliptic of t0 is subtracted from tropical positions referred to the ecliptic of date.

NOTE: this method will not provide accurate results if you want coordinates referred to the ecliptic of one of the following equinoxes:

```
#define SE_SIDM_J2000 18
#define SE_SIDM_J1900 19
#define SE_SIDM_B1950 20
```

Instead, you have to use a correct coordinate transformation as described in the following:

Special uses of the sidereal functions:

a) user-defined ayanamsha with t0 in UT.

If a user-defined ayanamsha is set using SE_SIDM_USER, then the t0 is usually considered to be TT (ET). However, t0 can be provided as UT if SE_SIDM_USER is combined with SE_SIDBIT_USER_UT.

/* with user-defined ayanamsha, t0 is UT */

#define SE_SIDBIT_USER_UT 1024

E.g.:

swe_set_sid_mode(SE_SIDM_USER + SE_SIDBIT_USER_UT, 1720935.589444445, 0);

iflag |= SEFLG_SIDEREAL;

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

iflgret = **swe_calc**(tjd, ipl, iflag, xp, serr);

printf("%f\n", xp[0]);

}

b) Transformation of ecliptic coordinates to the ecliptic of a > particular date. To understand these options, please study them in > the General Documentation of the Swiss Ephemeris (swisseph.html, > swisseph.pdf).

If a transformation to the **ecliptic of t0** is required the following bit can be added ('ored') to the value of the variable sid_mode:

/* for projection onto ecliptic of t0 */

#define SE_SIDBIT_ECL_T0 256

E.g.:

swe_set_sid_mode(SE_SIDM_J2000 + SE_SIDBIT_ECL_T0, 0, 0);

iflag |= SEFLG_SIDEREAL;

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

iflgret = **swe_calc**(tjd, ipl, iflag, xp, serr);

printf("%f\n", xp[0]);

}

This procedure is required for the following sidereal modes, i.e. for transformation to the ecliptic of one of the standard equinoxes:

#define SE_SIDM_J2000 18

#define SE_SIDM_J1900 19

#define SE_SIDM_B1950 20

If a transformation to the **ecliptic of date** is required the following bit can be added ('ored') to the value of the variable sid_mode:

/* for projection onto ecliptic of t0 */

#define SE_SIDBIT_ECL_DATE 2048

E.g.:

swe_set_sid_mode(SE_SIDM_J2000 + SE_SIDBIT_ECL_DATE, 0, 0);

iflag |= SEFLG_SIDEREAL;

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

iflgret = **swe_calc**(tjd, ipl, iflag, xp, serr);

printf("%f\n", xp[0]);

}

c) calculating precession-corrected transits.

The function **swe_set_sid_mode()** can also be used for calculating "precession-corrected transits". There are two methods, of which you have to choose the one that is more appropriate for you:

1. If you already have tropical positions of a natal chart, you can proceed as follows:

iflgret = **swe_calc**(tjd_et_natal, SE_ECL_NUT, 0, x, serr);

nut_long_natal = x[2];

swe_set_sid_mode(SE_SIDBIT_USER + SE_SIDBIT_ECL_T0, tjd_et, nut_long_natal);

where tjd_et_natal is the Julian day of the natal chart (Ephemeris time).

After this calculate the transits, using the function swe_calc() with the sidereal bit:

iflag |= SEFLG_SIDEREAL;

iflgret = **swe_calc**(tjd_et_transit, ipl_transit, iflag, xpt, serr);

2. If you do not have tropical natal positions yet, if you do not need them and are just interested in transit times, you can have it simpler:

swe_set_sid_mode(SE_SIDBIT_USER + SE_SIDBIT_ECL_T0, tjd_et, 0);

iflag |= SEFLG_SIDEREAL;

iflgret = **swe_calc**(tjd_et_natal, ipl_natal, iflag, xp, serr);

iflgret = **swe_calc**(tjd_et_transit, ipl_transit, iflag, xpt, serr);

In this case, the natal positions will be tropical but without nutation. Note that you should not use them for other purposes.

d) solar system rotation plane.

For sidereal positions referred to the solar system rotation plane, use the flag:

/* for projection onto solar system rotation plane */

#define SE_SIDBIT_SSY_PLANE 512

NOTE: the parameters set by **swe_set_sid_mode()** survive calls of the function **swe_close()**.

swe_get_ayanamsa_ex_ut(), swe_get_ayanamsa_ex(), swe_get_ayanamsa and swe_get_ayanamsa_ut()

These functions compute the ayanamsha, i.e. the distance of the tropical vernal point from the sidereal zero point of the zodiac. The ayanamsha is used to compute sidereal planetary positions from tropical ones:

pos_sid = pos_trop – ayanamsha

Important information concerning the values returned:

- The functions **swe_get_ayanamsa()** and **swe_get_ayanamsa_ut()** provide the ayanamsha without nutation.
- The functions **swe_get_ayanamsa_ex()** and **swe_get_ayanamsa_ex_ut()** provide the ayanamsha with or without nutation depending on the parameter iflag. If iflag contains (SEFLG_NONUT) the ayanamsha value is calculated without nutation, otherwise it is calculated including nutation.

It is **not** recommended to use the ayanamsha functions for calculating sidereal planetary positions from tropical positions, since this could lead to complicated confusions. For sidereal planets, please use **swe_calc_ut()** and **swe_calc()** with the flag SEFLG_SIDEREAL.

Use the ayanamsha function only for "academical" purposes, e.g. if you want to indicate the value of the ayanamsha on a horoscope chart. In this case, it is recommended to indicate the ayanamsha including nutation.

Ayanamsha without nutation may be useful in historical research, where the focus usually is on the mere precessional component of the ayanamsha.

Special case of "true" ayanamshas such as "True Chitrapaksha" etc.: The flags SEFLG_TRUEPOS, SEFLG_NOABERR and SEFLG_NOGDEFL can be used here, but users should not do that unless they really understand what they are doing. It means that the same flags are internally used for the calculation of the reference star (e.g. Citra/Spica). Slightly different ayanamsha values will result depending on these flags.

Before calling one of these functions, you have to set the sidereal mode with **swe_set_sid_mode**, unless you want the default sidereal mode, which is the Fagan/Bradley ayanamsha.

/* input variables:

* tjd_ut = Julian day number in UT

* (tjd_et = Julian day number in ET/TT)

* iflag = ephemeris flag (one of SEFLG_SWIEPH, SEFLG_JPLEPH, SEFLG_MOSEPH)

* plus some other optional SEFLG_...

* output values

* daya = ayanamsha value (pointer to double)

* serr = error message or warning (pointer to string)

* The function returns either the ephemeris flag used or ERR (-1)

*/

int32 **swe_get_ayanamsa_ex_ut**(

double tjd_ut,

int32 iflag,

double *daya,

char *serr);

int32 **swe_get_ayanamsa_ex**(

double tjd_et,

int32 iflag,

double *daya,

char *serr);

double **swe_get_ayanamsa_ut**(

double tjd_ut); /* input: Julian day number in UT */

double **swe_get_ayanamsa**(

double tjd_et); /* input: Julian day number in ET/TT */

The functions **swe_get_ayanamsa_ex_ut()** and **swe_get_ayanamsa_ex()** were introduced with Swiss Ephemeris version 2.02, the former expecting input time as UT, the latter as ET/TT.

This functions are **better** than the older functions **swe_get_ayanamsa_ut()** and **swe_get_ayanamsa()**.

The function **swe_get_ayanamsa_ex_ut()** uses a Delta T consistent with the ephe_flag specified.

The function **swe_get_ayanamsa_ex()** does not depend on Delta T; however with fixed-star-based ayanamshas like True Chitrapaksha or True Revati, the fixed star position also depends on the solar ephemeris (annual aberration of the star), which can be calculated with any of the three ephemeris flags.

The differences between the values provided by the new and old functions are **very small** and possibly only relevant for precision fanatics.

The function **swe_get_ayanamsa_ut()** was introduced with Swisseph Version 1.60 and expects Universal Time instead of Ephemeris Time. (cf. **swe_calc_ut()** and **swe_calc()**)

The Ephemeris file related functions (moved to 2.)

Information concerning the functions **swe_set_ephe_path()**, **swe_close()**, **swe_set_jpl_file()**, and **swe_version()** has been moved to **chapter 2**.

The sign of geographical longitudes in Swisseph functions

There is a disagreement between American and European programmers whether eastern or western geographical longitudes ought to be considered positive. Americans prefer to have West longitudes positive, Europeans prefer the older tradition that considers East longitudes as positive and West longitudes as negative.

The Astronomical Almanac still follows the European pattern. It gives the geographical coordinates of observatories in "East longitude".

The Swiss Ephemeris also **follows** the European style. All Swiss Ephemeris functions that use geographical coordinates consider **positive geographical longitudes as East** and **negative ones as West**.

E.g. 87w39 = -87.65° (Chicago IL/USA) and 8e33 = +8.55° (Zurich, Switzerland).

There is no such controversy about northern and southern geographical latitudes. North is always positive and south is negative.

Geographic versus geocentric latitude

There is some confusion among astrologers whether they should use geographic latitude (also called geodetic latitude, which is a synonym) or geocentric latitude for house calculations, topocentric positions of planets, eclipses, etc.

Where latitude is an input parameter (or output parameter) in Swiss Ephemeris functions, it is **always** geographic latitude. This is the latitude found in Atlases and Google Earth.

If internally in a function a conversion to geocentric latitude is required (because the 3-d point on the oblate Earth is needed), this is done automatically.

For such conversions, however, the Swiss Ephemeris only uses an ellipsoid for the form of the Earth. It does not use the irregular geoid. This can result in an altitude error of up to 500 meters, or error of the topocentric Moon of up to 0.3 arc seconds.

Astrologers who claim that for computing the ascendant or houses one needs geocentric latitude are wrong. The flattening of the Earth does not play a part in house calculations. Geographic latitude should **always** be used with house calculations.

House cusp calculation

swe_house_name()

/* returns the name of the house method, maximum 40 chars */
char ***swe_house_name**(
int hsys); /* house method, ascii code of one of the letters PKORCAEVXHTBG */

swe_houses()

/* house cusps, ascendant and MC */
int **swe_houses**(
    double tjd_ut, /* Julian day number, UT */
    double geolat, /* geographic latitude, in degrees */
    double geolon, /* geographic longitude, in degrees
    * eastern longitude is positive,
    * western longitude is negative,
    * northern latitude is positive,
    * southern latitude is negative */
    int hsys, /* house method, ascii code of one of the letters documented below */
    double *cusps, /* array for 13 (or 37 for hsys G) doubles, explained further below */
    double *ascmc); /* array for 10 doubles, explained further below */

swe_houses_armc() and swe_houses_armc_ex2()

int **swe_houses_armc**(
    double armc, /* ARMC */
    double geolat, /* geographic latitude, in degrees */
    double eps, /* ecliptic obliquity, in degrees */
    int hsys, /* house method, ascii code of one of the letters documented below */
    double *cusps, /* array for 13 (or 37 for hsys G) doubles, explained further below */
    double *ascmc); /* array for 10 doubles, explained further below */

int **swe_houses_armc_ex2**(
double armc, /* ARMC */
double geolat, /* geographic latitude, in degrees */
double eps, /* ecliptic obliquity, in degrees */
int hsys, /* house method, ascii code of one of the letters documented below */
double *cusps, /* array for 13 (or 37 for hsys G) doubles, explained further below */
double *ascmc, /* array for 10 doubles, explained further below */
double *cusp_speed,
double *ascmc_speed,
char *serr):

swe_houses_ex() and swe_houses_ex2()

/* extended function; to compute tropical or sidereal positions of house cusps */
int **swe_houses_ex**(
double tjd_ut, /* Julian day number, UT */
int32 iflag, /* 0 or SEFLG_SIDEREAL or SEFLG_RADIANS or SEFLG_NONUT */
double geolat, /* geographic latitude, in degrees */
double geolon, /* geographic longitude, in degrees
* eastern longitude is positive,
* western longitude is negative,
* northern latitude is positive,
* southern latitude is negative */
int hsys, /* house method, one-letter case sensitive code (list, see further below) */
double *cusps, /* array for 13 (or 37 for hsys G) doubles, explained further below */
double *ascmc); /* array for 10 doubles, explained further below */
/* extended function swe_houses_ex2():
* This function has the advantage that it also returns the speeds
* (daily motions) of the ascendant, midheaven and house cusps.
* In addition, it can return an error message or warning.
*/
int **swe_houses_ex2**(
double tjd_ut, /* Julian day number, UT */
int32 iflag, /* 0 or SEFLG_SIDEREAL or SEFLG_RADIANS or SEFLG_NONUT */
double geolat, /* geographic latitude, in degrees */
double geolon, /* geographic longitude, in degrees
* eastern longitude is positive,
* western longitude is negative,

* northern latitude is positive,

* southern latitude is negative */

int hsys, /* house method, one-letter case sensitive code (list, see further below) */

double *cusps, /* array for 13 (or 37 for hsys G) doubles, explained further below */

double *ascmc, /* array for 10 doubles, explained further below */

double *cusp_speed, /* like cusps */

double *ascmc_speed, /* like ascmc */

char *serr);

Note that all these functions tjd_ut must be Universal Time.

Also **note** that the array cusps must provide space for **13 doubles** (declare as cusp[13]), otherwise you risk a program crash. With house system 'G' (Gauquelin sector cusps), declare it as cusp[37].

With house system 'G', the cusp numbering is in clockwise direction.

The extended house functions **swe_houses_ex()** and **swe_houses_ex2()** do exactly the same calculations as **swe_houses()**. The difference is that the extended functions have a parameter iflag, which can be set to SEFLG_SIDEREAL, if sidereal house positions are wanted. The house function returns data based on the **true** equator and equinox of date. If the flag SEFLG_NONUT is set, then the house cusps will be based on the **mean** equator and equinox of date. However, we recommend to use the true equator and equinox. The function **swe_houses_ex2()** also provides the speeds ("daily motions") of the house cusps and additional points.

Before calling **swe_houses_ex()** or **swe_houses_ex()** for sidereal house positions, the sidereal mode can be set by calling the function **swe_set_sid_mode()**. If this is not done, the default sidereal mode, i.e. the Fagan/Bradley ayanamsha, will be used.

The function **swe_houses()**, **swe_houses_ex()**, and **swe_houses_ex2()** are most comfortable, as long as houses are to be calculated *for a given date and geographic position*. Sometimes, however, one will need to compute houses *from a given ARMC*, e.g. with the composite horoscope, which has no date, only a composite ARMC which is computed from two natal ARMCs. In this case, the function **swe_houses_armc()** or **swe_houses_armc_ex2()** can be used. Since these functions require the ecliptic obliquity eps, .anchor}one will probably want to calculate a composite value for this parameter also. To do this, one has to call **swe_calc()** with ipl = SE_ECL_NUT for both birth dates and then calculate the average of both eps.

"Sunshine" or Makransky houses require a special handling with the function **swe_houses_armc()** or **swe_houses_armc_ex2()**. The house system requires as a parameter the declination of the Sun. The user has to calculate the declination of the Sun and save it in the variable ascmc[9]. For house cusps of a composite chart, one has to calculate the composite declination of the Sun (= average of the declinations of the natal Suns).

There is no extended function for **swe_houses_armc()**. Therefore, if one wants to compute such exotic things as the house cusps of a sidereal composite chart, the procedure will be more complicated:

/* sidereal composite house computation; with true epsilon, but without nutation in longitude */

swe_calc_ut(tjd_ut1, SE_ECL_NUT, 0, x1, serr);

swe_calc_ut(tjd_ut2, SE_ECL_NUT, 0, x2, serr);

armc1 = **swe_sidtime**(tjd_ut1) * 15;

armc2 = **swe_sidtime**(tjd_ut2) * 15;

armc_comp = **composite**(armc1, armc2); /* this is a function created by the user */

eps_comp = (x1[0] + x2[0]) / 2;

// ayanamsha for the middle of the two birth days.

// alternatively, one could take the mean ayanamsha of the two birth dates.

// the difference will be microscopic.

tjd_comp = (tjd_ut1 + tjd_ut2) / 2;

retval = **swe_get_ayanamsa_ex_ut**(tjd_comp, iflag, &aya, serr);

swe_houses_armc(armc_comp, geolat, eps_comp, hsys, cusps, ascmc);

for (i = 1; i <= 12; i++)

cusp[i] = **swe_degnorm**(cusp[i] – aya);

for (i = 0; i < 10; i++)

ascmc[i] = **swe_degnorm**(asc_mc[i] – aya);

Or if you want to calculate sidereal progressions, do as follows:

- calculate the tropical radix_armc;
- radix_armc + direction_arc = directed_armc;
- use **swe_houses_armc**(directed_armc, ...) or **swe_houses_armc_ex2()** for the house cusps;
- subtract ayanamsha (**swe_get_ayanamsa_ex_ut()**) from the values.

Output and input parameters of the house function:

The first array element **cusps[0]** is always 0, the twelve houses follow **in cusps[1] .. [12]**, the reason being that arrays in C begin with the index 0. The indices are therefore:

cusps[0] = 0

cusps[1] = house 1

cusps[2] = house 2

etc.

In the array **ascmc**, the function returns the following values:

ascmc[0] = Ascendant

ascmc[1] = MC

ascmc[2] = ARMC

ascmc[3] = Vertex

ascmc[4] = ”equatorial ascendant”

ascmc[5] = ”co-ascendant” (Walter Koch)

ascmc[6] = ”co-ascendant” (Michael Munkasey)

ascmc[7] = ”polar ascendant” (M. Munkasey)

The following defines can be used to find these values:

```
#define SE_ASC 0
#define SE_MC 1
#define SE_ARMC 2
#define SE_VERTEX 3
#define SE_EQUASC 4 /* ”equatorial ascendant” */
#define SE_COASC1 5 /* ”co-ascendant” (W. Koch) */
#define SE_COASC2 6 /* ”co-ascendant” (M. Munkasey) */
#define SE_POLASC 7 /* ”polar ascendant” (M. Munkasey) */
```

#define SE_NASCMC 8

ascmc must be an array of **10 doubles**. **ascmc[8... 9]** are 0 and may be used for additional points in future releases.

The codes **hsys** of the most important house methods are:

hsys = 'P' Placidus

'K' Koch

'O' Porphyrius

'R' Regiomontanus

'C' Campanus

'A' or 'E' Equal (cusp 1 is Ascendant)

'W' Whole sign

The complete list of house methods in alphabetical order is:

hsys = 'B' Alcabitus

'Y' APC houses

'X' Axial rotation system / Meridian system / Zariel

'H' Azimuthal or horizontal system

'C' Campanus

'F' Carter "Poli-Equatorial"

'A' or 'E' Equal (cusp 1 is Ascendant)

'D' Equal MC (cusp 10 is MC)

'N' Equal/1=Aries

'G' Gauquelin sector

Goelzer -> Krusinski

Horizontal system -> Azimuthal system

'I' Sunshine (Makransky, solution Treindl)

'i' Sunshine (Makransky, solution Makransky)

'K' Koch

'U' Krusinski-Pisa-Goelzer

Meridian system -> axial rotation

'M' Morinus

Neo-Porphyry -> Pullen SD

Pisa -> Krusinski

'P' Placidus

Poli-Equatorial -> Carter

'T' Polich/Page ("topocentric" system)

'O' Porphyrius

'L' Pullen SD (sinusoidal delta) – ex Neo-Porphyry

'Q' Pullen SR (sinusoidal ratio)

'R' Regiomontanus

'S' Sripati

"Topocentric" system -> Polich/Page

'V' Vehlow equal (Asc. in middle of house 1)

'W' Whole sign

Zariel -> Axial rotation system

Placidus and Koch house cusps as well as Gauquelin sectors **cannot be computed beyond the polar circle**. In such cases, **swe_houses()** switches to Porphyry houses (each quadrant is divided into three equal parts) and returns the error code ERR. In addition, Sunshine houses may fail, e.g. when required for a date which is outside the time range of our solar ephemeris. Here, also, Porphyry houses will be provided.

The house method codes are actually case sensitive. At the moment, there still are no lowercase house method codes, and if a lowercase code is given to the function, it will be converted to uppercase. However, in future releases, lower case codes may be used for new house methods. In such cases, lower and uppercase won't be equivalent anymore.

The **Vertex** is the point on the ecliptic that is located in precise western direction. The opposition of the **Vertex** is the **Antivertex,** the ecliptic east point.

House position of a planet: swe_house_pos()

To compute the house position of a given body for a given ARMC, you may use:

double **swe_house_pos**(

double armc, /* ARMC */

double geolat, /* geographic latitude, in degrees */

double eps, /* ecliptic obliquity, in degrees */

int hsys, /* house method, one of the letters PKRCAV */

double *xpin, /* array of 2 doubles: ecl. longitude and latitude of the planet */

char *serr); /* return area for error or warning message */

The variables **armc**, **geolat**, **eps,** and **xpin[0]** and **xpin[1]** (ecliptic longitude and latitude of the planet) must be in degrees. **serr** must, as usually, point to a character array of 256 byte.

The function returns a value between 1.0 and 12.999999, indicating in which house a planet is and how far from its cusp it is.

With house system 'G' (Gauquelin sectors), a value between 1.0 and 36.9999999 is returned. Note that, while all other house systems number house cusps in counterclockwise direction, Gauquelin sectors are numbered in clockwise direction.

With Koch houses, the function sometimes returns 0, if the computation was not possible. This happens most often in polar regions, but it can happen at latitudes **below 66°33'** as well, e.g. if a body has a high declination and falls within the circumpolar sky. With circumpolar fixed stars (or asteroids) a Koch house position may be impossible at any geographic location except on the equator.

The user must decide how to deal with this situation.

You can use the house positions returned by this function for house horoscopes (or "mundane" positions). For this, you have to transform it into a value between 0 and 360 degrees. Subtract 1 from the house number and multiply it with 30, or mund_pos = (hpos – 1) * 30.

You will realize that house positions computed like this, e.g. for the Koch houses, will not agree exactly with the ones that you get applying the Huber "hand calculation" method. If you want a better agreement, set the ecliptic latitude **xpin[1] = 0**. Remaining differences result from the fact that Huber's hand calculation is a simplification, whereas our computation is geometrically accurate.

Currently, geometrically correct house positions are provided for the following house methods:

P Placidus, K Koch, C Campanus, R Regiomontanus, U Krusinski,

A/E Equal, V Vehlow, W Whole Signs, D Equal/MC, N Equal/Zodiac,

O Porphyry, B Alcabitius, X Meridian, F Carter, M Morinus,

T Polich/Page, H Horizon, G Gauquelin.

A simplified house position (distance_from_cusp / house_size) is currently provided for the following house methods:

Y APC houses, L Pullen SD, Q Pullen SR, I Sunshine, S Sripati.

This function requires TROPICAL positions in xpin. SIDEREAL house positions are identical to tropical ones in the following cases:

- If the traditional method is used to compute sidereal planets (sid_pos = trop_pos – ayanamsha). Here the function **swe_house_pos()** works for all house systems.
- If a non-traditional method (projection to the ecliptic of t0 or to the solar system rotation plane) is used and the definition of the house system does not depend on the ecliptic. This is the case with Campanus, Regiomontanus, Placidus, Azimuth houses, axial rotation houses. This is **not** the case with equal houses, Porphyry and Koch houses. You have to compute equal and Porphyry house positions on your own. **We recommend to avoid Koch** houses here. Sidereal Koch houses make no sense with these sidereal algorithms.

Calculating the Gauquelin sector position of a planet with swe_house_pos() or swe_gauquelin_sector()

For general information on Gauquelin sectors, read chapter 6.5 in documentation file swisseph.doc.

There are two functions that can be used to calculate Gauquelin sectors:

- **swe_house_pos**. Full details about this function are presented in the previous section. To calculate Gauquelin sectors the parameter hsys must be set to 'G' (Gauquelin sectors). This function will then return the sector position as a value between 1.0 and 36.9999999. Note that Gauquelin sectors are numbered in clockwise direction, unlike all other house systems.
- **swe_gauquelin_sector** - detailed below.

Function **swe_gauquelin_sector()** is declared as follows:

int32 **swe_gauquelin_sector**(

double tjd_ut, /* input time (UT) */

int32 ipl, /* planet number, if planet, or moon

* ipl is ignored if the following parameter (starname) is set */

char *starname, /* star name, if star */

int32 iflag, /* flag for ephemeris and SEFLG_TOPOCTR */

int32 imeth, /* method: 0 = with lat., 1 = without lat.,

* 2 = from rise/set, 3 = from rise/set with refraction */

double *geopos, /* array of three doubles containing

* geograph. long., lat., height of observer */

double atpress, /* atmospheric pressure, only useful with imeth = 3;

* if 0, default = 1013.25 mbar is used*/

double attemp, /* atmospheric temperature in degrees Celsius, only useful with imeth = 3 */

double *dgsect, /* return address for Gauquelin sector position */

char *serr); /* return address for error message */

This function returns OK or ERR (-1). It returns an error in a number of cases, for example circumpolar bodies with imeth=2. As with other SE functions, if there is an error, an error message is written to serr. dgsect is used to obtain the Gauquelin sector position as a value between 1.0 and 36.9999999. Gauquelin sectors are numbered in clockwise direction.

There are six methods of computing the Gauquelin sector position of a planet:

1. Sector positions from ecliptical longitude AND latitude:

There are two ways of doing this:

- Call **swe_house_pos()** with hsys = 'G', xpin[0] = ecliptical longitude of planet, and xpin[1] = ecliptical latitude. This function returns the sector position as a value between 1.0 and 36.9999999.
- Call **swe_gauquelin_sector()** with imeth = 0. This is less efficient than swe_house_pos because it recalculates the whole planet whereas swe_house_pos() has an input array for ecliptical positions calculated before.

2. Sector positions computed from ecliptical longitudes without ecliptical latitudes:

There are two ways of doing this:

- Call **swe_house_pos()** with hsys = 'G', xpin[0] = ecl. longitude of planet, and xpin[1] = 0. This function returns the sector position as a value between 1.0 and 36.9999999.
- Call **swe_gauquelin_sector()** with imeth = 1. Again this is less efficient than **swe_house_pos**.

3. Sector positions of a planet from rising and setting times of planets.

The rising and setting of the disk center is used:

- Call **swe_gauquelin_sector()** with imeth = 2.

4. Sector positions of a planet from rising and setting times of planets, taking into account atmospheric refraction.

The rising and setting of the disk center is used:

- Call **swe_gauquelin_sector()** with imeth = 3.

5. Sector positions of a planet from rising and setting times of planets.

The rising and setting of the disk edge is used:

- Call **swe_gauquelin_sector()** with imeth = 4.

6. Sector positions of a planet from rising and setting times of planets, taking into account atmospheric refraction.

The rising and setting of the disk edge is used:

- Call **swe_gauquelin_sector()** with imeth = 5.

Sidereal time with swe_sidtime() and swe_sidtime0()

The sidereal time is computed inside the **houses()** function and returned via the variable armc which measures sidereal time in degrees. To get sidereal time in hours, divide armc by 15.

If the sidereal time is required separately from house calculation, two functions are available. The second version requires obliquity and nutation to be given in the function call, the first function computes them internally. Both return sidereal time at the Greenwich Meridian, measured in hours.

double **swe_sidtime**(

    double tjd_ut); /* Julian day number, UT */

double **swe_sidtime0**(

    double tjd_ut, /* Julian day number, UT */

    double eps, /* obliquity of ecliptic, in degrees */

    double nut); /* nutation in longitude, in degrees */

Summary of SWISSEPH functions

Calculation of planets and stars

Planets, moon, asteroids, lunar nodes, apogees, fictitious bodies

```
// planetary positions from UT

int32 swe_calc_ut(
  double tjd_ut, /* Julian day number, Universal Time */
  int32 ipl, /* planet number */
  int32 iflag, /* flag bits */
  double *xx, /* target address for 6 position values: longitude,
          latitude, distance, * long. speed, lat. speed, dist. speed */
  char *serr /* 256 bytes for error string */
);

// planetary positions from TT

int32 swe_calc(
  double tjd_et, /* Julian day number, Ephemeris Time */
  int32 ipl, /* planet number */
  int32 iflag, /* flag bits */
  double *xx, /* target address for 6 position values: longitude,
  latitude, distance, *long. speed, lat. speed, dist. speed */
  char *serr /* 256 bytes for error string */
);

// planetary positions, planetocentric, from TT

int32 swe_calc_pctr(
  double tjd, // input julian day number in TT
  int32 ipl, // target object
  int32 iplctr, // center object
  int32 iflag, /* flag bits, as with swe_calc() */
  double *xxret,
  char *serr
);

// positions of planetary nodes and aspides from UT

int32 swe_nod_aps_ut(
  double tjd_ut, /* Julian day number, Universal Time */
```

```
  int32 ipl, /* planet number */
  int32 iflag, /* flag bits */
  int32 method, /* method SE_NODBIT_... (see docu above) */
  double *xnasc, /* target address for 6 position values for ascending node (cf. swe_calc()*/
  double *xndsc, /* target address for 6 position values for descending node (cf. swe_calc()*/
  double *xperi, /* target address for 6 position values for perihelion (cf. swe_calc()*/
  double *xaphe, /* target address for 6 position values for aphelion (cf. swe_calc()*/
  char *serr
);

// positions of planetary nodes and aspides from TT

int32 swe_nod_aps(
double tjd_et, /* Julian day number, Ephemeris Time */
int32 ipl, /* planet number */
int32 iflag, /* flag bits */
int32 method, /* method SE_NODBIT_... (see docu above) */
double *xnasc, /* target address for 6 position values for ascending node (cf. swe_calc()*/
double *xndsc, /* target address for 6 position values for descending node (cf. swe_calc()*/
double *xperi, /* target address for 6 position values for perihelion (cf. swe_calc()*/
double *xaphe, /* target address for 6 position values for aphelion (cf. swe_calc()*/
char *serr
);
```

Set the geographic location for topocentric planet computation

```
void swe_set_topo(
  double geolon, /* geographic longitude */
  double geolat, /* geographic latitude
          * eastern longitude is positive,
          * western longitude is negative,
          * northern latitude is positive,
          * southern latitude is negative */
  double altitude /* altitude above sea */
);
```

Set the sidereal mode and get ayanamsha values

```
void swe_set_sid_mode(
  int32 sid_mode,
  double t0, /* reference epoch */
  double ayan_t0 /* initial ayanamsha at t0 */
);

/* The function calculates ayanamsha for a given date in UT.
* The return value is either the ephemeris flag used or ERR (-1) */

int32 swe_get_ayanamsa_ex_ut(
  double tjd_ut, /* Julian day number in UT */
  int32 ephe_flag, /* ephemeris flag, one of SEFLG_SWIEPH, SEFLG_JPLEPH, SEFLG_MOSEPH */
  double *daya, /* output: ayanamsha value (pointer to double) */
  char *serr /* output: error message or warning (pointer to string) */
);
```

```
/* The function calculates ayanamsha for a given date in ET/TT.
* The return value is either the ephemeris flag used or ERR (-1) */

int32 swe_get_ayanamsa_ex(
  double tjd_ut, /* Julian day number in ET/TT */
  int32 ephe_flag, /* ephemeris flag, one of SEFLG_SWIEPH, SEFLG_JPLEPH, SEFLG_MOSEPH */
  double *daya, /* output: ayanamsha value (pointer to double) */
  char *serr /* output: error message or warning (pointer to string) */
);

/* to get the ayanamsha for a date in UT, old function, better use
swe_get_ayanamsa_ex_ut() */

double swe_get_ayanamsa_ut(double tjd_ut);

/* to get the ayanamsha for a date in ET/TT, old function, better use
swe_get_ayanamsa_ex() */

double swe_get_ayanamsa(double tjd_et);

// find the name of an ayanamsha

const char *swe_get_ayanamsa_name(int32 isidmode)
```

Eclipses and planetary phenomena

Find the next eclipse for a given geographic position

```
int32 swe_sol_eclipse_when_loc(
  double tjd_start, /* start date for search, Jul. day UT */
  int32 ifl, /* ephemeris flag */
  double *geopos, /* 3 doubles for geo. lon, lat, height */
  double *tret, /* return array, 10 doubles, see below */
  double *attr, /* return array, 20 doubles, see below */
  AS_BOOL backward, /* TRUE, if backward search */
  char *serr /* return error string */
);
```

Find the next eclipse globally

```
int32 swe_sol_eclipse_when_glob(
  double tjd_start, /* start date for search, Jul. day UT */
  int32 ifl, /* ephemeris flag */
  int32 ifltype, /* eclipse type wanted: SE_ECL_TOTAL etc. */
  double *tret, /* return array, 10 doubles, see below */
  AS_BOOL backward, /* TRUE, if backward search */
  char *serr /* return error string */
);
```

Compute the attributes of a solar eclipse for a given tjd, geographic long., latit. and height

```
int32 swe_sol_eclipse_how(
  double tjd_ut, /* time, Jul. day UT */
  int32 ifl, /* ephemeris flag */
  double *geopos, /* geogr. longitude, latitude, height */
  double *attr, /* return array, 20 doubles, see below */
  char *serr /* return error string */
);
```

Find out the geographic position where a central eclipse is central or a non-central one maximal

```
int32 swe_sol_eclipse_where(
  double tjd_ut, /* time, Jul. day UT */
  int32 ifl, /* ephemeris flag */
  double *geopos, /* return array, 10 doubles, geo. long. and lat. */
  double *attr, /* return array, 20 doubles, see below */
  char *serr /* return error string */
);
```

or

```
int32 swe_lun_occult_where(
  double tjd_ut, /* time, Jul. day UT */
  int32 ipl, /* planet number */
  char* starname, /* star name, must be NULL or "" if not a star */
  int32 ifl, /* ephemeris flag */
  double *geopos, /* return array, 10 doubles, geo. long. and lat. */
  double *attr, /* return array, 20 doubles, see below */
  char *serr /* return error string */
);
```

Find the next occultation of a body by the moon for a given geographic position

(can also be used for solar eclipses)

```
int32 swe_lun_occult_when_loc(
  double tjd_start, /* start date for search, Jul. day UT */
  int32 ipl, /* planet number */
  char* starname, /* star name, must be NULL or "" if not a star */
  int32 ifl, /* ephemeris flag */
  double *geopos, /* 3 doubles for geo. lon, lat, height */
  double *tret, /* return array, 10 doubles, see below */
  double *attr, /* return array, 20 doubles, see below */
  AS_BOOL backward, /* TRUE, if backward search */
  char *serr /* return error string */
);
```

Find the next occultation globally

(can also be used for solar eclipses)

```
int32 swe_lun_occult_when_glob(
  double tjd_start, /* start date for search, Jul. day UT */
  int32 ipl, /* planet number */
  char* starname, /* star name, must be NULL or "" if not a star */
  int32 ifl, /* ephemeris flag */
  int32 ifltype, /* eclipse type wanted */
  double *tret, /* return array, 10 doubles, see below */
  AS_BOOL backward, /* TRUE, if backward search */
  char *serr /* return error string */
);
```

Find the next lunar eclipse observable from a geographic location

```
int32 swe_lun_eclipse_when_loc(
  double tjd_start, /* start date for search, Jul. day UT */
  int32 ifl, /* ephemeris flag */
  double *geopos, /* 3 doubles for geo. lon, lat, height */
  double *tret, /* return array, 10 doubles, see below */
  double *attr, /* return array, 20 doubles, see below */
  AS_BOOL backward, /* TRUE, if backward search */
  char *serr /* return error string */
);
```

Find the next lunar eclipse, global function

```
int32 swe_lun_eclipse_when(
  double tjd_start, /* start date for search, Jul. day UT */
  int32 ifl, /* ephemeris flag */
  int32 ifltype, /* eclipse type wanted: SE_ECL_TOTAL etc. */
  double *tret, /* return array, 10 doubles, see below */
  AS_BOOL backward, /* TRUE, if backward search */
  char *serr /* return error string */
);
```

Compute the attributes of a lunar eclipse at a given time

```
int32 swe_lun_eclipse_how(
  double tjd_ut, /* time, Jul. day UT */
  int32 ifl, /* ephemeris flag */
  double *geopos, /* input array, geopos, geolon, geoheight */
  double *attr, /* return array, 20 doubles, see below */
  char *serr /* return error string */
);
```

Compute risings, settings and meridian transits of a body

```
int32 swe_rise_trans(
  double tjd_ut, /* search after this time (UT) */
  int32 ipl, /* planet number, if planet or moon */
  char *starname, /* star name, if star */
  int32 epheflag, /* ephemeris flag */
```

```
  int32 rsmi, /* integer specifying that rise, set, or one of the two meridian transits is wanted. see d
  double *geopos, /* array of three doubles containing geograph.  long., lat., height of observer */
  double atpress, /* atmospheric pressure in mbar/hPa */
  double attemp, /* atmospheric temperature in deg. C */
  double *tret, /* return address (double) for rise time etc. */
  char *serr /* return address for error message */
);

int32 swe_rise_trans_true_hor(
  double tjd_ut, /* search after this time (UT) */
  int32 ipl, /* planet number, if planet or moon */
  char *starname, /* star name, if star */
  int32 epheflag, /* ephemeris flag */
  int32 rsmi, /* integer specifying that rise, set, or one of the two meridian transits is wanted. see d
  double *geopos, /* array of three doubles containing * geograph. long., lat., height of observer */
  double atpress, /* atmospheric pressure in mbar/hPa */
  double attemp, /* atmospheric temperature in deg. C */
  double horhgt, /* height of local horizon in deg at the point where the body rises or sets*/
  double *tret, /* return address (double) for rise time etc. */
  char *serr /* return address for error message */
);
```

Compute heliacal risings and settings and related phenomena

```
int32 swe_heliacal_ut(
  double tjdstart, /* Julian day number of start date for the search of the heliacal event */
  double *dgeo /* geographic position (details below) */
  double *datm, /* atmospheric conditions (details below) */
  double *dobs, /* observer description (details below) */
  char *objectname, /* name string of fixed star or planet */
  int32 event_type, /* event type (details below) */
  int32 helflag, /* calculation flag, bitmap (details below) */
  double *dret, /* result: array of at least 50 doubles, of which 3 are used at the moment */
  char * serr /* error string */
);

// details of heliacal risings/settings

double swe_heliacal_pheno_ut(
  double tjd_ut, /* Julian day number */
  double *dgeo, /* geographic position (details under swe_heliacal_ut() */
  double *datm, /* atmospheric conditions (details under swe_heliacal_ut()) */
  double *dobs, /* observer description (details under swe_heliacal_ut()) */
  char *objectname, /* name string of fixed star or planet */
  int32 event_type, /* event type (details under function swe_heliacal_ut()) */
  int32 helflag, /* calculation flag, bitmap (details under swe_heliacal_ut()) */
  double *darr, /* return array, declare array of 50 doubles */
  char *serr /* error string */
);

// magnitude limit for visibility

double swe_vis_limit_mag(
```

```
  double tjdut, /* Julian day number */
  double *dgeo /* geographic position (details under swe_heliacal_ut() */
  double *datm, /* atmospheric conditions (details under swe_heliacal_ut()) */
  double *dobs, /* observer description (details under swe_heliacal_ut()) */
  char *objectname, /* name string of fixed star or planet */
  int32 helflag, /* calculation flag, bitmap (details under swe_heliacal_ut()) */
  double *dret, /* result: magnitude required of the object to be visible */
  char * serr /* error string */
);

double swe_heliacal_pheno_ut(
  double tjd_ut, /* Julian day number */
  double *dgeo, /* geographic position (details under swe_heliacal_ut() */
  double *datm, /* atmospheric conditions (details under swe_heliacal_ut()) */
  double *dobs, /* observer description (details under swe_heliacal_ut()) */
  char *objectname, /* name string of fixed star or planet */
  int32 event_type, /* event type (details under function swe_heliacal_ut()) */
  int32 helflag, /* calculation flag, bitmap (details under swe_heliacal_ut()) */
  double *darr, /* return array, declare array of 50 doubles */
  char *serr /* error string */
);
```

Compute planetary phenomena

```
int32 swe_pheno_ut(
  double tjd_ut, /* time Jul. Day UT */
  int32 ipl, /* planet number */
  int32 iflag, /* ephemeris flag */
  double *attr, /* return array, 20 doubles, see below */
  char *serr /* return error string */
);

int32 swe_pheno(
  double tjd_et, /* time Jul. Day ET */
  int32 ipl, /* planet number */
  int32 iflag, /* ephemeris flag */
  double *attr, /* return array, 20 doubles, see below */
  char *serr /* return error string */
);
```

Compute azimuth/altitude from ecliptic or equator

```
void swe_azalt(
  double tjd_ut, /* UT */
  int32 calc_flag, /* SE_ECL2HOR or SE_EQU2HOR */
  double *geopos, /* array of 3 doubles: geogr. long., lat., height */
  double atpress, /* atmospheric pressure in mbar (hPa) */
  double attemp, /* atmospheric temperature in degrees Celsius */
  double *xin, /* array of 3 doubles: position of body in either
          ecliptical or equatorial coordinates, depending on calc_flag */
  double *xaz /* return array of 3 doubles, containing azimuth, true altitude, apparent altitude */
);
```

Compute ecliptic or equatorial positions from azimuth/altitude

```
void swe_azalt_rev(
  double tjd_ut,
  int32 calc_flag, /* either SE_HOR2ECL or SE_HOR2EQU */
  double *geopos, /* array of 3 doubles for geograph. pos. of observer */
  double *xin, /* array of 2 doubles for azimuth and true altitude of planet */
  double *xout /* return array of 2 doubles for either ecliptic or
      equatorial coordinates, depending on calc_flag */
);
```

Compute refracted altitude from true altitude or reverse

```
double swe_refrac(
  double inalt,
  double atpress, /* atmospheric pressure in mbar (hPa) */
  double attemp, /* atmospheric temperature in degrees Celsius */
  int32 calc_flag /* either SE_TRUE_TO_APP or SE_APP_TO_TRUE */
);

double swe_refrac_extended(
  double inalt, /* altitude of object above geometric horizon in
  degrees, where geometric horizon = plane perpendicular to gravity */
  double geoalt, /* altitude of observer above sea level in meters */
  double atpress, /* atmospheric pressure in mbar (hPa) */
  double lapse_rate, /* (dattemp/dgeoalt) = [°K/m] */
  double attemp, /* atmospheric temperature in degrees Celsius */
  int32 calc_flag, /* either SE_TRUE_TO_APP or SE_APP_TO_TRUE */
  double *dret /* array of 4 doubles; declare 20 ! */
);
```

- – dret[0] true altitude, if possible; otherwise input value
- – dret[1] apparent altitude, if possible; otherwise input value
- – dret[2] refraction
- – dret[3] dip of the horizon

Compute Kepler orbital elements of a planet or asteroid

```
int32 swe_get_orbital_elements(
  double tjd_et, // input date in TT (Julian day number)
  int32 ipl, // planet number
  int32 iflag, // flag bits, see detailed docu
  double *dret, // return values, see detailed docu
  char *serr
);
```

Compute maximum/minimum/current distance of a planet or asteroid

Date and time conversion

Delta T from Julian day number

```
/* Ephemeris time (ET) = Universal time (UT) + swe_deltat_ex(UT) */

double swe_deltat_ex(
  double tjd, /* Julian day number in ET/TT */
  int32 ephe_flag, /* ephemeris flag (one of SEFLG_SWIEPH, SEFLG_JPLEPH, SEFLG_MOSEPH) */
  char *serr /* error message or warning */
);
```

older function:

```
double swe_deltat( double tjd);
```

Julian day number from year, month, day, hour, with check whether date is legal

```
/* Return value: OK or ERR */

int swe_date_conversion(
  int y, int m, int d, /* year, month, day */
  double hour, /* hours (decimal, with fraction) */
  char c, /* calendar 'g'[regorian] | 'j'[ulian] */
  double *tjd /* target address for Julian day */
);
```

Julian day number from year, month, day, hour

```
double swe_julday(
  int year,
  int month,
  int day,
  double hour,
  int gregflag /* Gregorian calendar: 1, Julian calendar: 0 */
);
```

Year, month, day, hour from Julian day number

```
void swe_revjul(
  double tjd, /* Julian day number */
  int gregflag, /* Gregorian calendar: 1, Julian calendar: 0 */
  int *year, /* target addresses for year, etc. */
  int *month,
  int *day,
  double *hour
);
```

Local time to UTC and UTC to local time

```
/* transform local time to UTC or UTC to local time
* input:
* iyear ... dsec date and time
* d_timezone timezone offset
* output:
* iyear_out ... dsec_out
*
* For time zones east of Greenwich, d_timezone is positive.
* For time zones west of Greenwich, d_timezone is negative.
*
* For conversion from local time to utc, use +d_timezone.
* For conversion from utc to local time, use -d_timezone.
*/

void swe_utc_timezone(
  int32 iyear, int32 imonth, int32 iday,
  int32 ihour, int32 imin, double dsec,
  double d_timezone,
  int32 *iyear_out, int32 *imonth_out, int32 *iday_out,
  int32 *ihour_out, int32 *imin_out, double *dsec_out
);
```

UTC to jd (TT and UT1)

```
/* input: date and time (wall clock time), calendar flag.
* output: an array of doubles with Julian Day number in ET (TT) and UT (UT1)
* an error message (on error)
* The function returns OK or ERR.
*/

void swe_utc_to_jd(
  int32 iyear, int32 imonth, int32 iday,
  int32 ihour, int32 imin, double dsec, /* NOTE: second is a decimal */
  gregflag, /* Gregorian calendar: 1, Julian calendar: 0 */
  dret /* return array, two doubles:
    * dret[0] = Julian day in ET (TT)
    * dret[1] = Julian day in UT (UT1) */
  serr /* error string */
);
```

TT (ET1) to UTC

```
/* input: Julian day number in ET (TT), calendar flag
* output: year, month, day, hour, min, sec in UTC */

void swe_jdet_to_utc(
  double tjd_et, /* Julian day number in ET (TT) */
  gregflag, /* Gregorian calendar: 1, Julian calendar: 0 */
  int32 *iyear, int32 *imonth, int32 *iday,
  int32 *ihour, int32 *imin, double *dsec /* NOTE: second is a decimal */
);
```

UT (UT1) to UTC

```
/* input: Julian day number in UT (UT1), calendar flag
* output: year, month, day, hour, min, sec in UTC */

void swe_jdut1_to_utc(
  double tjd_ut, // Julian day number in UT (UTC)
  gregflag,      // Gregorian calendar: 1, Julian calendar: 0
  int32 *iyear,
  int32 *imonth,
  int32 *iday,
  int32 *ihour,
  int32 *imin,
  double *dsec  // NOTE: second is a decimal
);
```

Get tidal acceleration used in swe_deltat()

```
double swe_get_tid_acc(void);
```

Set tidal acceleration to be used in swe_deltat()

```
void swe_set_tid_acc(double t_acc);
```

Equation of time

```
/* function returns the difference between local apparent and local mean time.
e = LAT -- LMT. tjd_et is ephemeris time */

int swe_time_equ(
  double tjd_et,
  double *e,
  char *serr
);

/* converts Local Mean Time (LMT) to Local Apparent Time (LAT) */

/* tjd_lmt and tjd_lat are a Julian day number
* geolon is geographic longitude, where eastern
* longitudes are positive, western ones negative */

int32 swe_lmt_to_lat(
  double tjd_lmt,
  double geolon,
  double *tjd_lat,
  char *serr
);

/* converts Local Apparent Time (LAT) to Local Mean Time (LMT) */

int32 swe_lat_to_lmt(
  double tjd_lat,
```

```
  double geolon,
  double *tjd_lmt,
  char *serr
);
```

Initialization, setup, and closing functions

Set directory path of ephemeris files

```
void swe_set_ephe_path(char *path);

/* set name of JPL ephemeris file */

void swe_set_jpl_file(char *fname);

/* close Swiss Ephemeris */

void swe_close(void);

/* find out version number of your Swiss Ephemeris version */

char *swe_version(char *svers);

/* svers is a string variable with sufficient space to contain the version number (255 char) */

/* find out the library path of the DLL or executable */

char *swe_get_library_path(char *spath);

/* spath is a string variable with sufficient space to contain the library path (255 char) */

/* find out start and end date of *se1 ephemeris file after a call of swe_calc() */

const char *CALL_CONV swe_get_current_file_data(
  int ifno,
  double *tfstart,
  double *tfend,
  int *denum
);
```

House calculation

Sidereal time

```
double swe_sidtime(double tjd_ut); /* Julian day number, UT */

double swe_sidtime0(
  mouble tjd_ut,    // Julian day number, UT
  double eps,       // obliquity of ecliptic, in degrees
  double nut        // nutation, in degrees
);
```

Name of a house method

```
char *swe_house_name(
  int hsys // house method, ascii code of one of the letters PKORCAEVXHTBG..
);
```

House cusps, ascendant and MC

```
int swe_houses(
  double tjd_ut, // Julian day number, UT
  double geolat, // geographic latitude, in degrees
  double geolon, // geographic longitude, in degrees,
        // eastern longitude is positive,
        // western longitude is negative,
        // northern latitude is positive,
        // southern latitude is negative
  int hsys, // house method, one of the letters PKRCAV
  double *cusps, // array for 13 doubles
  double *ascmc // array for 10 doubles
);
```

Extended house function; to compute tropical or sidereal positions

```
int swe_houses_ex(
  double tjd_ut,    // Julian day number, UT
  int32 iflag,      // 0 or SEFLG_SIDEREAL or SEFLG_RADIANS
  double geolat,    // geographic latitude, in degrees
  double geolon,    // geographic longitude, in degrees
  int hsys,         // house method, one of the hsys letters PKRCAV...
  double* cusps,    // array for 13 doubles
  double* ascmc     // array for 10 doubles
);

int swe_houses_ex2(
double tjd_ut,      // Julian day number, UT
int32 iflag,        // 0 or SEFLG_SIDEREAL or SEFLG_RADIANS or SEFLG_NONUT
double geolat,      // geographic latitude, in degrees
double geolon,      // geographic longitude, in degrees
int hsys,       // house method, one-letter case sensitive code
double *cusps,      // array for 13 (or 37 for system G) doubles
double *ascmc,      // array for 10 doubles
double *cusp_speed,     // like cusps
double *ascmc_speed,    // like ascmc
char *serr
);

int swe_houses_armc(
double armc,    // ARMC
double geolat,  // geographic latitude, in degrees
double eps,     // ecliptic obliquity, in degrees
int hsys,   // house method, one of the letters PKRCAV
double *cusps,  // array for 13 doubles
double *ascmc   // array for 10 doubles
```

```
);

int swe_houses_armc_ex2(
double armc,    // ARMC
double geolat,  // geographic latitude, in degrees
double eps,     // ecliptic obliquity, in degrees
int hsys,   // house method, ascii code of one of the hsys letters
double *cusps,  // array for 13 (or 37 for system G) doubles., explained further below
double *ascmc,  // array for 10 doubles
double *cusp_speed,
double *ascmc_speed,
char *serr
):
```

Get the house position of a celestial point

```
double swe_house_pos(
  double armc,      // ARMC
  double geolat,    // geographic latitude, in degrees
  double eps,       // ecliptic obliquity, in degrees
  int hsys,         // house method, one of the letters PKRCAV
  double *xpin,     // array of 2 doubles: ecl. longitude and latitude of the planet
  char *serr        // return area for error or warning message
);
```

Get the Gauquelin sector position for a body

```
double swe_gauquelin_sector(
  double tjd_ut,    // search after this time (UT)
  int32 ipl,        // planet number, if planet, or moon
  char *starname,   // star name, if star
  int32 iflag,      // flag for ephemeris and SEFLG_TOPOCTR
  int32 imeth,      // method: 0 = with lat., 1 = without lat.,
            // 2 = from rise/set, 3 = from rise/set with refraction
  double *geopos,   // array of three doubles containing
            // geograph. long., lat., height of observer
  double atpress,   //  atmospheric pressure, only useful with imeth = 3;
            //  if 0, default = 1013.25 mbar is used
  double attemp,    // atmospheric temperature in degrees Celsius, only useful with imeth = 3
  double *dgsect,   // return address for Gauquelin sector position
  char *serr        //  return address for error message
);
```

Auxiliary functions

swe_cotrans(): coordinate transformation, from ecliptic to equator or vice-versa

```
/* equator -> ecliptic : eps must be positive
* ecliptic -> equator : eps must be negative
* eps, longitude and latitude are in positive degrees! */
```

```
void swe_cotrans(
double *xpo,    // 3 doubles: long., lat., dist. to be converted;
        // distance remains unchanged, can be set to 1.00
double *xpn,    // 3 doubles: long., lat., dist. Result of the conversion
double eps  // obliquity of ecliptic, in degrees.
);
```

swe_cotrans_sp(): coordinate transformation of position and speed, from ecliptic to equator or vice-versa

- equator -> ecliptic : eps must be positive
- ecliptic -> equator : eps must be negative
- eps, long., lat., and speeds in long. and lat. are in degrees!

```
void swe_cotrans_sp(

double *xpo, /* 6 doubles, input: long., lat., dist. and speeds in
long., lat and dist. */

double *xpn, /* 6 doubles, position and speed in new coordinate
system */

double eps /* obliquity of ecliptic, in degrees. */
);
```

swe_get_planet_name(): get the name of a planet

```
char* swe_get_planet_name(
int ipl,    // planet number
char* plan_name // address for planet name, at least 20 char
);
```

swe_degnorm(): normalize degrees to the range 0 ... 360

```
double swe_degnorm(double x);
```

swe_radnorm(): normalize radians to the range 0 ... 2 PI

```
double swe_radnorm(double x);
```

swe_split_deg(): split degrees to sign/nakshatra, degrees, minutes, seconds of arc

This function takes a decimal degree number as input and provides sign or nakshatra, degree, minutes, seconds and fraction of second. It can also round to seconds, minutes, degrees. It can also split the value into zodiacal signs or Nakshatras.

```
void swe_split_deg(
  double ddeg,
  int32 roundflag,
  int32 *ideg,
```

```
  int32 *imin,
  int32 *isec,
  double *dsecfr,
  int32 *isgn
);
```

input:

```
ddeg          decimal degrees, ecliptic longitude
roundflag   by default there is no rounding. if rounding is required, the following bits can be set:
  # define SE_SPLIT_DEG_ROUND_SEC 1
  # define SE_SPLIT_DEG_ROUND_MIN 2
  # define SE_SPLIT_DEG_ROUND_DEG 4
  # define SE_SPLIT_DEG_ZODIACAL 8  // split into zodiac signs
  # define SE_SPLIT_DEG_NAKSHATRA 1024  // split into nakshatras
  # define SE_SPLIT_DEG_KEEP_SIGN 16    // don't round to next zodiac sign/nakshatra,
  # define SE_SPLIT_DEG_KEEP_DEG 32     // don't round to next degree
```

e.g. 29.9999998 will be rounded to 29°59’59” (or 29°59’ or 29°)

or next nakshatra: e.g. 13.3333332 will be rounded to 13°19’59” (or 13°19’ or 13°)

e.g. 10.9999999 will be rounded to 10d59’59” (or 10d59’ or 10d)

output:

```
ideg degrees,
imin minutes,
isec seconds,
dsecfr fraction of seconds
isgn    if SE_SPLIT_DEG_ZODIACAL    zodiac sign number 0..11
    if SE_SPLIT_DEG_NAKSHATRA   nakshatra number 0..26
    else +1 or -1
```

Other functions that may be useful

PLACALC, the predecessor of SWISSEPH, had included several functions that we do not need for SWISSEPH anymore. Nevertheless we include them again in the library, because some users of our software may have taken them over and use them in their applications. However, we gave them new names that were more consistent with SWISSEPH.

PLACALC used angular measurements in centiseconds a lot; a centisecond is **1/100** of an arc second. The C type CSEC or centisec is a 32-bit integer. CSEC was used because calculation with integer variables was considerably faster than floating point calculation on most CPUs in 1988, when PLACALC was written.

In the Swiss Ephemeris we have dropped the use of centiseconds and use double (64-bit floating point) for all angular measurements.

Normalize argument into interval [0..DEG360]

```
centisec swe_csnorm(centisec p);
```

Distance in centisecs p1 - p2 normalized to [0..360]

```
centisec swe_difcsn(centisec p1, centisec p2);
```

Distance in degrees

```
double swe_difdegn(double p1, double p2);
```

Distance in centisecs p1 - p2 normalized to [-180..180]

```
centisec swe_difcs2n(centisec p1, centisec p2);
```

Distance in degrees

```
double swe_difdeg2n(double p1, double p2);
```

Round second, but at 29.5959 always down

```
centisec swe_csroundsec(centisec x);
```

Double to int32 with rounding, no overflow check

```
int32 swe_d2l(double x);
```

Day of week

```
// Monday = 0, \... Sunday = 6,
int swe_day_of_week(double jd);
```

Centiseconds -> time string

```
char *swe_cs2timestr(CSEC t, int sep, AS_BOOL suppressZero, char *a);
```

Centiseconds -> longitude or latitude string

```
char *swe_cs2lonlatstr(CSEC t, char pchar, char mchar, char *s);
```

Centiseconds -> degrees string

```
char *swe_cs2degstr(CSEC t, char *a);
```

The SWISSEPH DLLs for Windows

There is a 32 bit Windows DLL swedll32.dll, and a 64 bit DLL swedll64.dl.

You can use our programs swetest.c and swewin.c as examples. To compile swetest or swewin with a DLL:

1. The compiler needs the following files:

```
swetest.c or swewin.c
swedll32.dll
swedll32.lib (if you choose implicit linking)
swephexp.h
swedll.h
sweodef.h
```

2. Define the following macros (-d):

```
USE_DLL
```

3. Build swetest.exe from swetest.c and swedll32.lib or swedll64.lib (depending on the 32-bit or 64-bit architecture of your system).

Build swewin.exe from swewin.c, swewin.rc, and swedll32.lib or swedll64.lib.

We provide some project files which we have used to build our test samples. You will need to adjust the project files to your environment.

We have worked with Microsoft Visual C++ 5.0 (32-bit). The DLLs where built with the Microsoft compilers.

Using the DLL with Visual Basic 5.0

(deprecated and not supported by Astrodienst)

The 32-bit DLL contains the exported function under 'decorated names'. Each function has an underscore before its name, and a suffix of the form @xx where xx is the number of stack bytes used by the call.

The Visual Basic declarations for the DLL functions and for some important flag parameters are in the file \sweph\vb\swedecl.txt and can be inserted directly into a VB program.

A sample VB program vbsweph is included on the distribution, in directory \sweph\vb. To run this sample, the DLL file swedll32.dll must be copied into the vb directory or installed in the Windows system directory.

DLL functions returning a string:

Some DLL functions return a string, e.g.

char* **swe_get_planet_name**(int ipl, char *plname)

This function copies its result into the string pointer plname; the calling program must provide sufficient space so that the result string fits into it. As usual in C programming, the function copies the return string into the provided area and returns the pointer to this area as the function value. This allows to use this function directly in a C print statement.

In VB there are three problems with this type of function:

1. The string parameter plname must be initialized to a string of sufficient length before the call; the content does not matter because it is overwritten by the called function. The parameter type must be

ByVal plname as String.

2. The returned string is terminated by a NULL character. This must be searched in VB and the VB string length must be set accordingly. Our sample program demonstrates how this can be done:

```
Private Function set_strlen(c$) As String
i = InStr(c$, Chr$(0))
c$ = Left(c$, i - 1)
set_strlen = c$
End Function
plname = String(20,0) ' initialize string to length 20
swe_get_planet_name(SE_SUN, plname)
plname = set_strlen(plname)
```

3. The function value itself is a pointer to character. This function value cannot be used in VB because VB does not have a pointer data type. In VB, such a Function can be either declared as type "As long" and the return value ignored, or it can be declared as a Sub. We have chosen to declare all such functions as ‚Sub', which automatically ignores the return value.

```
Declare Sub swe_get_planet_name(ByVal ipl as Long, ByVal plname as String).
```

Using the Swiss Ephemeris with different programming languages

Perl

The Swiss Ephemeris can be run from Perl using the Perl module SwissEph.pm. The module SwissEph.pm uses XSUB ("eXternal SUBroutine"), which calls the Swiss Ephemeris functions either from a C library or a DLL.

See Github repository https://github.com/aloistr/perl-sweph

PHP

See Github repository https://github.com/cyjoelchen/php-sweph

Python

See Github repository https://github.com/astrorigin/pyswisseph

(not officially supported by Astrodienst)

Java

See http://www.th-mack.de/download/swisseph-doc/swisseph/SwissEph.html

(not officially supported by Astrodienst)

Swisseph with different hardware and compilers

Depending on what hardware and compiler you use, there will be slight differences in your planetary calculations. For positions in longitude, they will be never larger than **0.0001"** in longitude. Speeds show no difference larger than **0.0002 arcsec/day.**

Some differences originate from the fact that the floating point arithmetic in Intel processor is executed with 80 bit precision, whereas stored program variables have only 64 bit precision. When code is optimized, more intermediate results are kept inside the processor registers, i.e. they are not shortened from 80bit to 64 bit. When these results are used for the next calculation, the outcome is then slightly different.

In the computation of speed for the nodes and apogee, differences between positions at close intervals are involved; the subtraction of nearly equal values results shows differences in internal precision more easily than other types of calculations. As these differences have no effect on any imaginable application software and are mostly within the design limit of Swiss Ephemeris, they can be safely ignored.

Debugging and Tracing Swisseph

This feature is deprecated. Sine release 2.10.02 no trace DLLs are provided. Trace code will be removed in one of the next releases.

If you are using the DLL

Besides the ordinary Swisseph function, there are two additional DLLs that allow you tracing your Swisseph function calls:

Swedlltrs32.dll and swedlltrs64.dll are for single task debugging, i.e. if only one application at a time calls Swisseph functions.

Two output files are written:

a) swetrace.txt: reports all Swisseph functions that are being called.

b) swetrace.c: contains C code equivalent to the Swisseph calls that your application did.

The last bracket of the function main() at the end of the file is missing.

If you want to compile the code, you have to add it manually. Note that these files may grow very fast, depending on what you are doing in your application. The output is limited to 10000 function calls per run.

Swedlltrm32.dll and swedlltrm64.dll are for multitasking, i.e. if more than one application at a time are calling Swisseph functions. If you used the single task DLL here, all applications would try to write their trace output into the same file. Swedlltrm32.dll and swedlltrm64.dll generate output file names that contain the process identification number of the application by which the DLL is called, e.g. swetrace_192.c and swetrace_192.txt.

Keep in mind that every process creates its own output files and with time might fill your disk.

In order to use a trace DLL, you have to replace your Swisseph DLL by it:

a) save your Swisseph DLL;

b) rename the trace DLL as your Swisseph DLL (e.g. as swedll32.dll or swedll64.dll).

```
Output samples swetrace.txt:

swe_deltat: 2451337.870000 0.000757

swe_set_ephe_path: path_in = path_set = \\sweph\\ephe\\

swe_calc: 2451337.870757 -1 258 23.437404 23.439365 -0.003530 -0.001961 0.000000 0.000000

swe_deltat: 2451337.870000 0.000757

swe_sidtime0: 2451337.870000 sidt = 1.966683 eps = 23.437404 nut = -0.003530
```

```
swe_sidtime: 2451337.870000 1.966683

swe_calc: 2451337.870757 0 258 77.142261 -0.000071 1.014989 0.956743 -0.000022 0.000132

swe_get_planet_name: 0 Sun
```

swetrace.c:

```
#include \"sweodef.h\"
#include \"swephexp.h\"
void main()
{
  double tjd, t, nut, eps; int i, ipl, retc; long iflag;
  double armc, geolat, cusp[12], ascmc[10]; int hsys;
  double xx[6]; long iflgret;
  char s[AS_MAXCH], star[AS_MAXCH], serr[AS_MAXCH];
  /*SWE_DELTAT*/
  tjd = 2451337.870000000;
  t = swe_deltat(tjd);
  printf("swe_deltat: %f\t%f\t\n", tjd, t);
  /*SWE_CALC*/
  tjd = 2451337.870757482; ipl = 0; iflag = 258;
  iflgret = **swe_calc**(tjd, ipl, iflag, xx, serr); /* xx = 1239992 */
  /*SWE_CLOSE*/
  swe_close();
}
```

If you are using the source code

Similar tracing is also possible if you compile the Swisseph source code into your application. Use the preprocessor definitions TRACE = 1 for single task debugging, and TRACE = 2 for multitasking. In most compilers this flag can be set with – DTRACE = 1 or / DTRACE = 1.

Updates

Updates of documention

when	by	what
30-sep-1997	Alois	added chapter 10 (sample programs)
7-oct-1997	Dieter	inserted chapter 7 (house calculation)
8-oct-1997	Dieter	appendix “Changes from version 1.00 to 1.01”
12-oct-1997	Alois	added new chapter 10 Using the DLL with Visual Basic
26-oct-1997	Alois	improved implementation and documentation of swe_fixstar()
28-oct-1997	Dieter	changes from Version 1.02 to 1.03
29-oct-1997	Alois	added VB sample extension, fixed VB declaration errors
9-nov-1997	Alois	added Delphi declaration sample
8-dec-1997	Dieter	remarks concerning computation of asteroids, changes to version 1.04
8-jan-1998	Dieter	changes from version 1.04 to 1.10.
12-jan-1998	Dieter	changes from version 1.10 to 1.11.
21-jan-1998	Dieter	calculation of topocentric planets and house positions (1.20)
28-jan-1998	Dieter	Delphi 1.0 sample and declarations for 16- and 32-bit Delphi (1.21)
11-feb-1998	Dieter	version 1.23
7-mar-1998	Alois	version 1.24 support for Borland C++ Builder added
4-jun-1998	Alois	version 1.25 sample for Borland Delphi-2 added
29-nov-1998	Alois	version 1.26 source code information added §16, Placalc API added
1-dec-1998	Dieter	chapter 19 and some additions in beginning of Appendix.
2-dec-1998	Alois	equation of time explained (in §4), changes version 1.27 explained
3-dec-1998	Dieter	note on ephemerides of 1992 QB1 and 1996 TL66
17-dec-1998	Alois	note on extended time range of 10’800 years

when	by	what
22-dec-1998	Alois	appendix A
12-jan-1999	Dieter	eclipse functions added, version 1.31
19-apr-1999	Dieter	version 1.4
8-jun-1999	Dieter	chapter 21 on tracing an debugging Swisseph
27-jul-1999	Dieter	info about sidereal calculations
16-aug-1999	Dieter	version 1.51, minor bug fixes
15-feb-2000	Dieter	many things for version 1.60
19-mar-2000	Vic Ogi	swephprg.doc re-edited
17-apr-2002	Dieter	documentation for version 1.64
26-jun-2002	Dieter	version 1.64.01
31-dec-2002	Alois	edited doc to remove references to 16-bit version
12-jun-2003	Alois/Dieter	documentation for version 1.65
10-jul-2003	Dieter	documentation for version 1.66
25-may-2004	Dieter	documentation of eclipse functions updated
31-mar-2005	Dieter	documentation for version 1.67
3-may-2005	Dieter	documentation for version 1.67.01
22-feb-2006	Dieter	documentation for version 1.70.00
2-may-2006	Dieter	documentation for version 1.70.01
5-feb-2006	Dieter	documentation for version 1.70.02
30-jun-2006	Dieter	documentation for version 1.70.03
28-sep-2006	Dieter	documentation for version 1.71
29-may-2008	Dieter	documentation for version 1.73
18-jun-2008	Dieter	documentation for version 1.74
27-aug-2008	Dieter	documentation for version 1.75
7-apr-2009	Dieter	documentation of version 1.76
3-sep-2013	Dieter	documentation of version 1.80
10-sep-2013	Dieter	documentation of version 1.80 corrected
11-feb-2014	Dieter	documentation of version 2.00
4-mar-2014	Dieter	documentation of swe_rise_trans() corrected
18-mar-2015	Dieter	documentation of version 2.01
11-aug-2015	Dieter	documentation of version 2.02
14-aug-2015	Dieter	documentation of version 2.02.01
16-oct-2015	Dieter	documentation of version 2.03
21-oct-2015	Dieter	documentation of version 2.04
27-may-2015	Dieter	documentation of version 2.05
27-may-2015	Dieter	documentation of version 2.05.01
10-jan-2016	Dieter	documentation of version 2.06
5-jan-2018	Dieter	documentation of version 2.07
1-feb-2018	Dieter	documentation of version 2.07.01
22-feb-2018	Dieter	docu of swe_fixstar2() improved
11-sep-2019	Simon Hren	Reformatting of documentation
22-jul-2020	Dieter	Documentation of version 2.09
23-jul-2020	Dieter	Documentation of version 2.09.01
27-jul-2020	Dieter	Small corrections
18-aug-2020	Dieter	Documentation of version 2.09.02
1-sep-2020	Dieter	Documentation of version 2.09.03s

when	by	what
1-dec-2020	Dieter	Documentation of version 2.10
9-dec-2020	Dieter	Dieter: “AD” replaced by “CE” and “BC” replaced by “BCE”.
15-dec-2020	Alois	Minor cosmetics
27-jan-2021	Dieter	Small additions and minor cosmetics based on proposals by Aum Hren
2-may-2021	Dieter	Release notes for version 2.10.01
4-aug-2021	Alois	Release notes for version 2.10.02
27-aug-2022	Alois	Release notes for version 2.10.03

Release History

Changes from version 2.10.02 to 2.10.03

A bug was fixed in function swe_lun_eclipse_when() which had lead to missing lunar eclipses between the years 776 and 967 CE.

The calculation of Moon‘s magnitude for large phase angles (when Moon is close to Sun) has been improved.

Changes from version 2.10.01 to 2.10.02

New functions were added, to find crossings of planets over fixed positions:

- swe_solcross()
- swe_mooncross()
- swe_mooncross_node()
- swe_helio_cross()

Changes from version 2.10 to 2.10.01

1. An old bug in the lunar ephemeris with (iflag & SEFLG_SWIEPH) was fixed. It resulted from the fact that a different ecliptic obliquity J2000 was used in the packing and unpacking of the ephemeris data (function sweph.c:rot_back()). The difference between the two was 0.042”. Many thanks to Hal Rollins for finding and reporting this problem.
2. Correct handling of new JPL Ephemeris DE441, using the lunar tidal acceleration (deceleration) -25.936 ”/cent^2 (according to Jon Giorgini/JPL’s Horizons System).
3. Deltat T was updated for current years.
4. A minor bug in swe_calc_pctr() was fixed: The function did not work correctly with asteroids and (iflag & SEFLG_JPLEPH), resulting in the error message ”Ephmeris file seas_18.se1 is damaged (2)”.
5. Bug in swe_rise_set() with fixed stars reported by Ricardo Ric was fixed.

Changes from version 2.09.03 to 2.10

New features:

- ephemerides of center of body (COB) of planets

- ephemerides of some planetary moons
- planetocentric ephemerides using the function swe_calc_pctr()
- function swe_get_current_file_data() for time range of *.se1 ephemeris files.

Changes from version 2.09.02 to 2.09.03

Three minor bug fixes:

- An initialization *serr = '\0'; was missing in function swe_calc(), which could lead to crashes where error messages were written.
- Sidereal positions of asteroids were wrong with ayanamshas 9-16, 21-26, 37, 38, 41, 42. (Namely, all ayanamshas whose initial date is given in UT.)
- Asteroids with ipl > 10000 (SE_AST_OFFSET): calculating with several different ayanamshas after each other did not work properly.

Changes from version 2.09.01 to 2.09.02

New functions swe_houses_ex2() and swe_houses_armc_ex2() can calculate speeds ("daily motions") of house cusps and related points.

Changes from version 2.09 to 2.09.01

Bugfix for improved Placidus house cusps near polar circle.

Changes from version 2.08 to 2.09

This release provides new values for Delta T in 2020 and 2021, an improved calculation of Placidus house cusps near the polar circles, new magnitudes for the major planets, improved sidereal ephemerides, and a few new ayanamshas.

1. Our calculation of Placidus house positions did not provide greatest possible precision with high geographic latitudes (noticed by D. Senthilathiban). The improvement is documented in the General Documentation under 6.7. "Improvement of the Placidus house calculation in SE 2.09".
2. New magnitudes according to Mallama 2018 were implemented. The new values agree with JPL Horizons for all planets except Mars, Saturn, and Uranus. Deviations form Horizons are < 0.1m for Mars, < 0.02m for Saturn and < 0.03m for Uranus.
3. New values for Delta T have been added for 2020 and 2021 (the latter estimated).

Sidereal astrology:

A lot of work has been done for more correct calculation of ayanamshas.

4. Improved general documentation:

- theory of ayanamsha in general
- about Lahiri ayanamsha
- about ayanamsha data in IAE, IENA, RP

These parts of the documentation have been improved considerably. Important contributions were made by D. Senthilathiban and A.K. Kaul.

(Thank you very much, indeed!)

If questions arise concerning the reproducibility of ayanamsha values as given in IAE, IENA, or Rashtriya Panchang, please study Appendix E in the general documentation.

5. Small corrections were to some ayanamshas whose original definition was based on an old precession model such as Newcomb or IAU 1976:

ayanamsha correction prec. model

```
0 Fagan-Bradley 0.41256" Newcomb

1 Lahiri -0.13036" IAU 1976

3 Raman 0.82800" Newcomb

5 Krishnamurti 0.82800" Newcomb
```

6. Additional, very small, corrections were made with the follwoing ayanamshas:

- Fagan/Bradley (0): Initial date is Besselian, i.e. 2433282.42346 instead of 2433282.5.
- Lahiri (1): Correction for nutation on initial date was slightly improved in agreement with IAE 1985, namely nutation Wahr (1980) instead of nutation IAU2000B.
- DeLuce (2): DeLuce assumed zero ayanamsha at 1 Jan. 1 BCE, but used Newcomb precession to determine the ayanamsha for current epochs. The ayanamsha is now based on modern precession. The correction amounts to about 22".

7. New ayanamshas:

- Krishnamurti/Senthilathiban, SE_SIDM_KRISHNAMURTI_VP291 45
- Lahiri 1940, SE_SIDM_LAHIRI_1940 43
- Lahiri 1980, SE_SIDM_LAHIRI_VP285 44
- Lahiri ICRC SE_SIDM_LAHIRI_ICRC 46

The three additional Lahiri ayanamshas are not really important. They were needed for testing and understanding the history of this aynamasha, and for the same reason they should also be kept. Our hitherto Lahiri ayanamsha (SE_SIDM_LAHIRI = 1) is still the official Lahiri ayanamsha as used in Indian Astronomical Ephemeris (IAE) since 1985.

8. Option for ayanamsha calculation relative to ecliptic of date.

```
#define SE_SIDBIT_ECL_DATE 2048
```

With swetest, the option -sidbit2048 can be used. (To be used by those only who understand it.)

Other issues:

9. When house calculation fails (which can happen with Placidus, Gauquelin, Koch, Sunshine houses), then the house functions return error but nevertheless provide Porphyry house cusps. Until now, swetest did so silently, without any warning. It now writes a warning.
10. Bug fix in function swehouse.c:swe_house_pos(): Corrected double hcusp[36] to double hcusp[37].
11. Bug fix in function swe_refrac_extended(), calculating true altitude from apparent altitude (SE_APP_TO_TRUE): Function now correctly returns true altitude if apparent altitude is greater or equal to the dip of the horizon.
12. Bug fix in function swe_get_planet_name() when used for asteroids. If the file s*.se1 was older than 2005, then the function provided a name string beginning with "? ".
13. Behaviour of occultation functions with fixed stars: attr[0] and attr[2] (fraction of diameter or disk occulted by the Moon) now have the value 1 (in previous versions they had value 100). The new value is consistent with those given with occultations of planetary disks.

14. The function swe_calc() near its beginning set serr = ” in versions up to 2.08. This destroyed possible warnings written into it in the calling function swe_calc().
15. Perl-Swisseph:

- Functions swe_sol_eclipse_where(), swe_sol_eclipse_how(), swe_lun_eclipse_how() now provide Saros numbers, in the array attr as well as in the variables saros_series and saros_no.
- Functions swe_lun_eclipse_when() now also provide start and end times ecl_begin und ecl_end (as with sol_eclipse_when_glob()).

Changes from version 2.07.01 to 2.08

This release provides a number minor bug fixes and cleanups, an update for current Delta T, a few little improvements of swetest and three new ayanamshas.

Fixed star functions:

- Wrong distance values in the remote past or future were corrected.

Position values were not affected by this bug.

- Inaccurate speed values of fixed star functions were corrected.

The nutation component was missing.

- When sepl*/semo* are not installed, swe_fixstar2() now defaults to the Moshier ephemeris. With version 2.07*, it has returned error.
- Repeated call of swe_fixstar_mag() did not work correctly with SE 2.07*. Now it does.
- The AU constant has been updated to the current IAU standard. This change does not have any noticeable effect on planetary or star positions.

Ayanamshas:

- New ayanamshas were added:

SE_SIDM_GALCENT_COCHRANE (David Cochrane)

SE_SIDM_GALEQU_FIORENZA (Nick Anthony Fiorenza)

SE_SIDM_VALENS_MOON (Vettius Valens, 2nd century CE)

For information on these, please look them up in the general documentation of the Swiss Ephemeris.

- Kugler ayanamshas were corrected:

E = -3;22 in source corresponds ayanamsha ay = 5;40

E = -4;46 in source corresponds ayanamsha ay = 4;16

E = -5;37 in source corresponds ayanamsha ay = 3;25

(Nobody has noticed this error for 20 years.)

Other stuff:

- swe_houses_ex() now also understands iflag & SEFLG_NONUT. This could be relevant for the calculation of sidereal house cusps.
- swe_pheno() and swe_pheno_ut(): the functions now return the correct ephemeris flag.
- swe_split_deg() has had a problem if called with

SE_SPLIT_DEG_ROUND_SEC or SE_SPLIT_DEG_ZODIACAL:

Sometimes, it provided sign number 12 when a position was rounded to 360°. This was wrong because sign numbers are defined as 0 - 11. This is a very old bug. From now on, only sign numbers 0 - 11 can occur.

A similar error occurred with SE_SPLIT_DEG_ROUND_SEC and SE_SPLIT_DEG_NAKSHATRA, where only nakshatra numbers 0 - 26 should be returned, no 27.

- Macros EXP16, USE_DLL16 und MAKE_DLL16 for very old compilers were removed.

Improvements of swetest:

- With calculations depending on geographic positions such as risings and local eclipses, an output line indicating the geographic position has been added. Those who use swetest system calls in their software (which we actually do not recommend) should test if this does not create.
- The output header of swetest now shows both true and mean epsilon.
- swetest option -sidudef[jd,ay0,...] allows user-defined ayanamsha. For detailed info about this option call swetest -h.

All new DLLs and executables were created with Microsoft Visual Studio 2015 (version 14.), no longer with MinGW on Linux. The usage of MinGW since Swiss Ephemeris version 2.05 had caused difficult problems for some of our users. We hope that these problems will now disappear.

Changes from version 2.07 to 2.07.01

- Changes for compatibility with Microsoft Visual C. Affected functions are: swe_fixstar2(), swe_fixstar2_ut(), swe_fixstar2_mag().
- Minor bugfixes in the functions swe_fixstar_ut(), swe_fixstar2_ut() and swe_fixstar2(). In particular, calls of the _ut functions with sequential star numbers did not work properly. This was an older bug, introduced with version 2.02.01 (where it appeared in function swe_fixstar_ut()).
- Wrong leap second (20171231) removed from swedate.c. Affected functions were: swe_utc_to_jd(), swe_jdet_to_utc(), swe_jdut1_to_utc().

Changes from version 2.06 to 2.07

- Greatly enhanced performance of swe_rise_trans() with calculations of risings and settings of planets except for high geographic latitudes.
- New functions swe_fixstar2(), swe_fixstar2_ut(), and swe_fixstar2_mag() with greatly increased performance. Important additional remarks are given further below.
- Fixed stars data file sefstars.txt was updated with new data from SIMBAD database.
- swe_fixstar(): Distances (in AU) and daily motions of the stars have been added to the return array. The daily motions contain components of precession, nutation, aberration, parallax and the proper motions of the stars. The usage of correct fixed star distances leads to small changes in fixed star positions and calculations of occultations of stars by the Moon (in particular swe_lun_occult_when_glob()).

To transform the distances from AU into lightyears or parsec, please use the following defines, which are in swephexp.h:

#define AUNIT_TO_LIGHTYEAR (1.0/63241.077088071)

#define AUNIT_TO_PARSEC (1.0/206264.8062471)

- There was a bug with daily motions of planets in sidereal mode: They contained precession! (Nobody ever noticed or complained for almost 20 years!)
- In JPL Horizons mode, the Swiss Ephemeris now reproduces apparent position as provided by JPL Horizons with an accuracy of a few milliseconds of arc for its *whole time range*. Until SE 2.06 this has been possible only after 1800. Please note, this applies to JPL Horizons mode only (SEFLG_JPLHOR and SEFLG_JPLHOR_APPROX together with an original JPL ephemeris file; or swetest -jplhor, swetest

-jplhora). Our default astronomical methods are those of IERS Conventions 2010 and Astronomical Almanac, *not* those of JPL Horizons.

- After consulting with sidereal astrologers, we have changed the behavior of the function swe_get_ayanamsa_ex(). See programmer's documentation swephprg.htm, chap. 10.2. Note this change has no impact on the calculation of planetary positions, as long as you calculate them using the sidereal flag SEFLG_SIDEREAL.
- New ayanamsha added:

"Vedic" ayanamsha according to Sunil Sheoran (SE_SIDM_TRUE_SHEORAN)

It must be noted that in Sheoran's opinion 0 Aries = 3°20' Ashvini. The user has to carry the responsibility to correctly handle this problem. For calculating a planet's nakshatra position correctly, we recommend the use of the function swe_split_deg() with parameter roundflag |= SE_SPLIT_DEG_NAKSHATRA or roundflag |= 1024. This will handle Sheoran's ayanamsha correctly.

For more information about this and other ayanamshas, I refer to the general documentation chap. 2.7 or my article on ayanamshas here: https://www.astro.com/astrology/in_ayanamsha_e.htm

- Function swe_rise_trans() has two new flags:

SE_BIT_GEOCTR_NO_ECL_LAT 128 /* use geocentric (rather than topocentric) position of object and ignore its ecliptic latitude */

SE_BIT_HINDU_RISING /* calculate risings according to Hindu astrology */

- Of course, as usual, leap seconds and Delta T have been updated.
- Calculation of heliacal risings using swe_heliacal_ut() now also works with Bayer designations, with an initial comma, e.g. ",alTau".
- Problem left undone:

Janez Križaj noticed that in the remote past the ephemeris of the Sun has some unusual ecliptic latitude, which amounts to +-51 arcsec for the year -12998. This phenomenon is due to an intrinsic inaccuracy of the precession theory Vondrak 2011 and therefore we do not try to fix it. While the problem could be avoided by using some older precession theory such as Laskar 1986 or Owen 1990, we give preference to Vondrak 2011 because it is in very good agreement with precession IAU2006 for recent centuries. Also, the "problem" (a very small one) appears only in the very remote past, not in historical epochs.

Important additional information on the new function swe_fixstar2() and its derivatives with increased performance:

Some users had criticized that swe_fixstar() was very inefficient because it reopened and scanned the file sefstars.txt for each fixed star to be calculated. With version 2.07, the new function swe_fixstar2() reads the whole file the first time it is called and saves all stars in a sorted array of structs. Stars are searched in this list using the binary search function bsearch(). After a call of swe_close() the data will be lost. A new call of swe_fixstar2() will reload all stars from sefstars.txt.

The declaration of swe_fixstar2() is identical to old swe_fixstar(), but its behavior is slightly different:

Fixed stars can be searched by

- full traditional name
- Bayer/Flamsteed designation
- traditional name with wildcard character '%'

(With previous versions, search string "aldeb" provided the star Aldebaran. This does not work anymore. For abbreviated search strings, a '%' wildcard must, be added, e.g. "aldeb%".)

With the old swe_fixstar(), it was possible to use numbers as search keys. The function then returned the n-th star it found in the list. This functionality is still available in the new version of the function, but the

star numbering does no longer follow the order of the stars in the file, but the order of the sorted Bayer designations. Nevertheless this feature is very practical if one wants to create a list of all stars.

```
for i=1; i\<10000; i++) { // choose any number greater than number of lines (stars) in file
  sprintf(star, \"%d\", i);
  returncode = swe_fixstar2(star, tjd, \...);
  //... whatever you want to do with the star positions ...
  if (returncode == ERR) break;
}
```

Changes from version 2.05.01 to 2.06

New calculation of Delta T, according to:

Stephenson, F.R., Morrison, L.V., and Hohenkerk, C.Y., "Measurement of the Earth's Rotation: 720 BCE to CE 2015", published by Royal Society Proceedings A and available from their website at

http://rspa.royalsocietypublishing.org/content/472/2196/20160404

http://astro.ukho.gov.uk/nao/lvm/

http://astro.ukho.gov.uk/nao/lvm/Table-S15.txt

This publication provides algorithms for Delta T from 721 BCE to 2016 CE based on historical observations of eclipses and occultations, as well as a parabolic function for epochs beyond this time range.

The new Swiss Ephemeris uses these algorithms before 1 Dec. 1955 and then switches over to values provided by Astronomical Almanac 1986(etc.) pp. K8-K9 and values from IERS.

Delta T values from 1973 to today have been updated by values from IERS, with four-digit accuracy. Two small bugs that interpolates these tabulated data have been fixed. Changes in Delta T within this time range are smaller than 5 millisec. The accuracy possible with 1-year step width is about 0.05 sec. For better accuracy, we would have to implement a table of monthly or daily delta t values.

Time conversions from or to UTC take into account the leap second of 31 Dec 2016.

Minor bug fixes in heliacal functions. E.g., heliacal functions now work with ObjectName in uppercase or lowercase.

Function swe_house_pos() now provides geometrically correct planetary house positions also for the house methods I, Y, S (Sunshine, APC, Scripati).

House method N (1 = 0° Widder) did not work properly with some sidereal zodiac options.

swe_houses_ex() with sidereal flag and rarely used flags SE_SIDBIT_ECL_T0 or SE_SIDBIT_SSY_PLANE returned a wrong ARMC.

Better behavior of swetest -rise in polar regions.

swetest understands a new parameter -utcHH:MM:SS, where input time is understood as UTC (whereas -utHH:MM:SS understands it as UT1). Note: Output of dates is always in UT1.

About 110 fixed stars were added to file sefstars.txt.

Changes from version 2.05 to 2.05.01

Bug in new function **swe_orbit_max_min_true_distance()** has been fixed.

Changes from version 2.04 to 2.05

Starting with release 2.05, the special unit test system **setest** designed and developed by Rüdiger Plantiko is used by the developers. This improves the reliability of the code considerably and has led to the discovery of multiple bugs and inconsistencies.

Note: **setest** is not to be confused with **swetest**, the test command-line utility program.

Bug fixes and new features:

1. The **Fixed stars file sefstars.txt** was updated with new data from the Simbad Database. Some errors in the file were fixed.
2. **Topocentric positions** of planets: The value of speed was not very good. This problem was found by Igor "TomCat" Germanenko in March 2015. A more accurate calculation of speed from three positions has now been implemented.

In addition, topocentric positions had an error < 1 arcsec if the function swe_calc() was called without SEFLG_SPEED. This problem was found by Bernd Müller and has now been fixed.

3. **Initial calls of the Swiss Ephemeris**: Some problems were fixed which appeared when users did calculations without opening the Swiss, i.e. without calling the function **swe_set_ephe_path**().

NOTE: It is still strongly recommended to call this function in the beginning of an application in order to make sure that the results are always consistent.

4. **New function swe_get_orbital_elements()** calculates osculating Kepler elements and some other data for planets, Earth-Moon barycentre, Moon, and asteroids. The program swetest has a new option -orbel that displays these data.

New function **swe_orbit_max_min_true_distance()** provides maximum, minimum, and true distance of a planet, on the basis of its osculating ellipse. The program swetest, when called with the option -fq, displays a relative distance of a planet (0 is maximum distance, 1000 is minimum distance).

5. New house methods were added:

F - Carter poli-equatorial house system

D - Equal houses, where cusp 10 = MC

I - Sunshine

N - Equal houses, where cusp 1 = 0 Aries

L - Pullen SD (sinusoidal delta) = ex Neo-Porphyry

Q - Pullen SR (sinusoidal ratio)

S - Sripati

Note:

- Sunshine houses require some special handling with the functions **swe_houses_armc()** and **swe_house_pos()**. Detailed instructions are given in the Programmer's Manual.
- Until version 2.04, the function **swe_house_pos()** has provided Placidus positions for the APC method. From version 2.05 on, it provides APC positions, but using a simplified method, namely the position relative to the house cusp and the house size. This is not really in agreement with the geometry of the house system.
- The same simplified algorithm has been implemented for the following house methods: Y APC, I Sunshine, L Pullen SD, Q Pullen SR, S Sripati

We hope to implement correct geometrical algorithms with time.

Minor bugfixes with houses:

- APC houses had nan (not a number) values at geographic latitude 0.

- APC houses had inaccurate MC/IC at geographic latitude 90.
- Krusinski houses had wrong (opposite) house positions with function swe_house_pos() at geographic latitude 0.0.

6. Sidereal zodiac defined relative to UT or TT:

A problem found by Parashara Kumar with the ayanamsha functions: The function swe_get_ayanamsa() requires TT (ET), but some of the ayanamshas were internally defined relative to UT. Resulting error in ayanamsha were about 0.01 arcsec in 500 CE. The error for current dates is about 0.0001 arcsec.

The internal definitions of the ayanamshas has been changed and can be based either on UT or on TT.

Nothing changes for the user, except with user-defined ayanamshas. The t0 used in swe_set_sid_mode() is considered to be TT, except if the new bit flag SE_SIDBIT_USER_UT (1024) is or'ed to the parameter sid_mode.

7. **Ayanamshas:** Some ayanamshas were corrected:

- The "True Revati Ayanamsha" (No. 28) (had the star at 0 Aries instead of 29°50' Pisces.
- The Huber Babylonian ayanamsha (No. 12) has been wrong for many years by 6 arc min. This error was caused by wrong information in a publication by R. Mercier. The correction was made according to Huber's original publication. More information is given in the General Documentation of the Swiss Ephemeris.
- Ayanamsha having Galactic Centre at 0 Sagittarius (No. 17) has been changed to a "true" ayanamsha that has the GC always at 0 Sag.

In addition, the following ayanamshas have been added:

- Galactic ayanamsha (Gil Brand) SE_SIDM_GALCENT_RGBRAND
- Galactic alignment (Skydram/Mardyks) SE_SIDM_GALALIGN_MARDYKS
- Galactic equator (IAU 1958) SE_SIDM_GALEQU_IAU1958
- Galactic equator true/modern SE_SIDM_GALEQU_TRUE
- Galactic equator in middle of Mula SE_SIDM_GALEQU_MULA
- True Mula ayanamsha (Chandra Hari) SE_SIDM_TRUE_MULA
- Galactic centre middle Mula (Wilhelm) SE_SIDM_GALCENT_MULA_WILHELM
- Aryabhata 522 SE_SIDM_ARYABHATA_522
- Babylonian Britton SE_SIDM_BABYL_BRITTON

More information about these ayanamshas is given in the General Documentation of the Swiss Ephemeris.

8. **_TRUE_ ayanamshas algorithm** (True Chitra, True Revati, True Pushya, True Mula, Galactic/Gil Brand, Galactic/Wilhelm) always keep the intended longitude, with or without the following iflags: SEFLG_TRUEPOS, SEFLG_NOABERR, SEFLG_NOGDEFL.

So far, the True Chitra ayanamsha had Spica/Chitra at 180° exactly if the *apparent* position of the star was calculated, however not if the *true* position (without aberration/light deflection) was calculated. However, some people may find it more natural if the star's true position is exactly at 180°.

9. Occultation function swe_lun_occult_when_loc():

- Function did not correctly detect daytime occurrence with partial occultations (a rare phenomenon).
- Some rare occultation events were missed by the function.

As a result of the changes three are very small changes in the timings of the events.

- Occultation of fixed stars have provided four contacts instead of two. Now there are only two contacts.

10. **Magnitudes for Venus and Mercury** have been improved according to Hilten 2005.

The Swiss Ephemeris now provides the same magnitudes as JPL's Horizons System.

11. **Heliacal functions**: A few bugs discovered by Victor Reijs have been fixed, which however did not become apparent very often.
12. **User-defined Delta T**: For archeoastronomy (as suggested by Victor Reijs) a new function swe_set_delta_t_userdef() was created that allows the user to set a particular value for delta t.
13. **Function swe_nod_aps**(): a bug was fixed that occurred with calculations for the EMB.
14. **New function swe_get_library_path**(): The function returns the path in which the executable resides. If it is running with a DLL, then returns the path of the DLL.

Changes from version 2.03 to 2.04

The DLL of version 2.03 is not compatible with existing software. In all past versions, the function names in the DLL were "decorated" (i.e. they had an initial '_' and a final '@99'). However, version 2.03 had the function names "undecorated". This was a result of the removal of the PASCAL keyword from the function declarations. Because of this, the DLL was created with the ___cdecl calling convention whereas with the PASCAL keyword it had been created with the ___stdcall calling convention.

Since VBA requires ___stdcall, we return to ___stdcall and to decorated function names.

The macro PASCAL_CONV, which had been misleading, was renamed as CALL_CONV.

Changes from version 2.02.01 to 2.03

This is a minor release, mainly for those who wish a thread-safe Swiss Ephemeris. It was implemented according to the suggestions made by Rüdiger Plantico and Skylendar. Any errors might be Dieter Koch's fault. On our Linux system, at least, it seems to work.

However, it seems that that we cannot build a thread-safe DLL inhouse at the moment. If a group member could provide a thread-safe DLL, that could be added to the Swiss Ephemeris download area.

Other changes:

FAR, PASCAL, and EXP16 macros in function declarations were removed.

Minor bug fixes:

- swe_calc_ut(): With some nonsensical SEFLG_ combinations, such as a combination of several ephemeris flags, slightly inconsistent results were returned.
- swe_calc(planet) with SEFLG_JPLEPH: If the function was called with a JD beyond the ephemeris range, then a subsequent call of swe_calc(SE_SUN) for a valid JD would have provided wrong result. This was a very old bug, found by Anner van Hardenbroek.

Note, other issues that have been discussed recently or even longer ago had to be postponed.

Changes from version 2.02 to 2.02.01

- For better backward-compatibility with 2.0x, the behavior of the old Delta T function swe_deltat() has been modified as follows:

swe_deltat() assumes

SEFLG_JPLEPH, if a JPL file is open;

SEFLG_SWIEPH, otherwise.

Usually, this modification does not result in values different from those provided by former versions SE 2.00 and 2.01.

Note, SEFLG_MOSEPH is never assumed by swe_deltat(). For consistent handling of ephemeris-dependent Delta T, please use the new Delta T function swe_deltat_ex(). Or if you understand the lunar tidal acceleration problem, you can use swe_set_tid_acc() to define the value you want.

- With version 2.02, software that does not use swe_set_ephe_path() or swe_set_jpl_file() to initialize the Swiss Ephemeris may fail to calculate topocentric planets with swe_calc() or swe_calc_ut() (return value ERR). Version 2.02.01 is more tolerant again.
- Ayanamshas TRUE_REVATI, TRUE_PUSHYA now also work if not fixed stars file is found in the ephemeris path. With TRUE_CHITRA, this has been the case for longer.
- Bug fixed: since version 2.00, the sidereal modes TRUE_CHITRA, TRUE_REVATI, TRUE_PUSHYA provided wrong latitude and speed for the Sun.

Thanks to Thomas Mack for some contributions to this release.

Changes from version 2.01 to 2.02

Many thanks to all who have contributed bug reports, in particular Thomas Mack, Bernd Müller, and Anner van Hardenbroek.

Swiss Ephemeris 2.02 contains the following updates:

- A bug was fixed in sidereal time functions before 1850 and after 2050. The bug was a side effect of some other bug fix in Version 2.01. The error was smaller than 5 arc min for the whole time range of the ephemeris.

The bug also resulted in errors of similar size in azimuth calculations before 1850 and after 2050.

Moreover, the bug resulted in errors of a few milliarcseconds in topocentric planetary positions before 1850 and after 2050.

In addition, the timings of risings, settings, and local eclipses may be slightly affected, again only before 1850 and after 2050.

- A bug was fixed that sometimes resulted in a program crash when function calls with different ephemeris flags (SEFLG_JPLEPH, SEFLG_SWIEPH, and SEFLG_MOSEPH) were made in sequence.
- Delta T functions:
- New function swe_deltat_ex(tjd_ut, ephe_flag, serr), where ephe_flag is one of the following:

SEFLG_SWIEPH, SEFLG_JPLEPH, SEFLG_MOSEPH, and serr the usual string for error messages.

It is wise to use this new function instead of the old swe_deltat(), especially if one uses more than one ephemeris or wants to compare different ephemerides in UT.

Detailed explanations about this point are given further below in the general remark concerning Swiss Ephemeris 2.02 and above in chap. 8 (on Delta T functions).

- The old function swe_deltat() was slightly modified. It now assumes

SEFLG_JPLEPH, if a JPL file is open;

SEFLG_SWIEPH, if a Swiss Ephemeris sepl* or semo* file is found;

SEFLG_MOSEPH otherwise.

Usually, this modification does not result in values different from those provided by former versions SE 2.00 and 2.01.

- Ayanamsha functions:
- New functions swe_get_ayanamsa_ex(), swe_get_ayanamsa_ex_ut() had to be introduced for similar reasons as swe_deltat_ex(). However, differences are very small, especially for recent dates.

For detailed explanations about this point, see general remarks further below.

- The old function swe_get_ayanamsa() was modified in a similar way as swe_deltat().

Usually, this modification does not result in different results.

- Eclipse and occultation functions:
- Searches for non-existing events looped through the whole ephemeris.

With version 2.02, an error is returned instead.

- Simplified (less confusing) handling of search flag in functions swe_sol_eclipse_when_glob() and swe_lun_occult_when_glob() (of course backward compatible).
- fixed bug: swe_lun_occult_when_loc() has overlooked some eclipses in polar regions (bug introduced in Swiss Ephemeris 2.01)
- SEFLG_JPLHOR also works in combination with SEFLG_TOPOCTR

swetest:

- The parameter -at(pressure),(temperature) can also be used with calculation of risings and altitudes of planets.
- Some rounding errors in output were corrected.
- swemptab.c was renamed swemptab.h.
- Small correction with SEFLG_MOSEPH: frame bias was not correctly handled so far. Planetary positions change by less than 0.01 arcsec, which is far less than the inaccuracy of the Moshier ephemeris.

A general remark concerning Swiss Ephemeris 2.02:

Since Swiss Ephemeris 2.0, which can handle a wide variety of JPL ephemerides, old design deficiencies of some functions, in particular swe_deltat(), have become incommoding under certain circumstances. Problems may (although need not) have occurred when the user called swe_calc_ut() or swe_fixstar_ut() for the remote past or future or compared planetary positions calculated with different ephemeris flags (SEFLG_SWIEPH, SEFLG_JPLEPH, SEFLG_MOSEPH).

The problem is that the Delta T function actually needs to know what ephemeris is being used but does not have an input parameter ephemeris_flag. Since Swiss Ephemeris 2.00, the function swe_deltat() has therefore made a reasonable guess what kind of ephemeris was being used, depending on the last call of the function swe_set_ephe_path(). However, such guesses are not necessarily always correct, and the functions may have returned slightly inconsistent return values, depending on previous calculations made by the user. Although the resulting error will be always smaller than the inherent inaccuracy in historical observations, the design of the function swe_deltat() is obviously inappropriate.

A similar problem exists for the function swe_get_ayanamsa() although the possible inconsistencies are very small.

To remedy these problems, Swiss Ephemeris 2.02 introduces new functions for the calculation of Delta T and ayanamsha:

swe_deltat_ex(),
swe_get_ayanamsa_ex_ut(), and
swe_get_ayanamsa_ex()
(The latter is independent of Delta T, however some ayanamshas like True Chitrapaksha depend on a precise fixed star calculation, which requires a solar ephemeris for annual aberration. Therefore, an ephemeris flag is required.)

Of course, the old functions swe_deltat(), swe_get_ayanamsa(), and swe_get_ayanamsa_ut() are still supported and work without any problems as long as the user uses only one ephemeris flag and calls the function swe_set_ephe_path() (as well swe_set_jpl_file() if using SEFLG_JPLEPH) before calculating Delta T and planetary positions. Nevertheless, it is recommended to *use the new functions swe_deltat_ex(), swe_get_ayanamsa_ex(), and swe_get_ayanamsa_ex_ut()* in future projects.

Also, please note that if you calculate planets using swe_calc_ut(), and stars using swe_fixstar_ut(), you usually need not worry about Delta T and can avoid any such complications.

Changes from version 2.00 to 2.01

Many thanks to those who reported bugs or made valuable suggestions. And I apologize if I forgot to mention some name.

Note: Still unsolved is the problem with the lunar node with SEFLG_SWIEPH, discovered recently by Mihai (I don't know his full name).

- https://groups.yahoo.com/neo/groups/swisseph/conversations/topics/4829?reverse=1

This problem, which has existed "forever", is tricky and will take more time to solve.

Improvements and updates:

- Lunar tidal acceleration for DE431 was updated to -25.8 arcsec/cty^2.

IPN Progress Report 42-196, February 15, 2014, p. 15: W.M. Folkner & alii, "The Planetary and Lunar Ephemerides DE430 and DE431".

- leap seconds of 2012 and 2015 added. (Note, users can add future leap seconds themselves in file seleapsec.txt.
- New values for Delta T until 2015, updated estimations for coming years.
- #define NO_JPL was removed
- True Pushya paksha ayanamsha added, according to PVR Narasimha Rao.

Fixes for bugs introduced with major release 2.0:

- Topocentric speed of planets was buggy after 2050 and before 1850, which was particularly obvious with slow planets like Neptune or Pluto. (Thanks to Igor "TomCat" Germanenko for pointing out this bug.)

This was caused by the new (since 2.0) long-term algorithm for Sidereal Time, which interfered with the function swe_calc().

- Topocentric positions of the *Moon* after 2050 and before 1850 had an error of a few arc seconds, due to the same problem. With the Sun and the planets, the error was < 0.01 arcsec.
- Another small bug with topocentric positions was fixed that had existed since the first release of topocentric calculations, resulting in very small changes in position for the whole time range of the ephemeris.

Errors due to this bug were < 0.3 arcsec for the Moon and < 0.001" for other objects.

- A small bug in the new long-term algorithm for Sidereal Time, which is used before 1850 and after 2050, was fixed. The error due to this bug was < 0.1 degree for the whole ephemeris time range.
- Since Version 2.0, swe_set_tid_acc() did not work properly anymore, as a result of the new mechanism that chooses tidal acceleration depending on ephemeris. However, this function is not really needed anymore.
- Sidereal modes SE_SIDBIT_ECL_T0, SE_SIDBIT_SSY_PLANE did not work correctly anymore with ayanamshas other than Fagan/Bradley.
- Ephemeris time range was corrected for a few objects:

Chiron ephemeris range defined as 675 CE to 4650 CE.

Pholus ephemeris range defined as -2958 (2959 BCE) to 7309 CE.

Time range of interpolated lunar apside defined as -3000 (3001 BCE) to 3000 CE.

- Suggestion by Thomas Mack, concerning 32-bit systems:

"... #define _FILE_OFFSET_BITS 64

has to appear before(!) including the standard libraries. ... You then can compile even on 32 bit systems without any need for workarounds."

Fixes for other bugs (all very old):

- Function swe_lun_eclipse_when_loc(): From now on, an eclipse is considered locally visible if the whole lunar disk is above the local geometric horizon. In former versions, the function has returned incorrect data if the eclipse ended after the rising of the upper and the rising of the lower limb of the moon or if it began between the setting of the lower and the setting of the upper limb of the moon.
- The same applies for the function swe_sol_eclipse_when_loc(), which had a similar problem.
- Some solar and lunar eclipses were missing after the year 3000 CE.

The following functions were affected:

swe_lun_eclipse_when(), swe_sol_eclipse_when_glob(), swe_sol_eclipse_when_loc().

There was no such problem with the remote past, only with the remote future.

- Functions swe_lunar_occult_when_glob() and swe_lunar_occult_when_loc() were improved. A better handling of rare or impossible events was implemented, so that infinite loops are avoided. For usage of the function, see example in swetest.c and programmers docu. The flag SE_ECL_ONE_TRY must be used, and the return value checked, unless you are really sure that events do occur.
- swe_nod_aps() now understands iflag & SEFLG_RADIANS
- In swetest, are rounding bug in degrees, minutes, seconds fixed.

180.0000000000000 could have been printed as "179°59'59.1000".

Changes from version 1.80 to 2.00

This is a major release which makes the Swiss Ephemeris fully compatible with JPL Ephemeris DE430/DE431.

A considerable number of functions were updated. That should not be a problem for existing applications. However, the following notes must be made:

1. New ephemeris files sepl*.se1 and semo*.se1 were created from DE431, covering the time range from 11 Aug. -12999 Jul. (= 4 May -12999 Greg.) to 7 Jan. 16800. For consistent ephemerides, **users are advised to use either old sepl* and semo* files (based on DE406) or new files (based on DE431) but not mix old and new ones together**. The internal handling of old and new files is not 100% identical (because of 3. below).
2. Because the time range of DE431 is a lot greater than that of DE406, better algorithms had to be implemented for objects not contained in JPL ephemerides (mean lunar node and apogee). Also, sidereal time and the equation of time had to be updated in order to give sensible results for the whole time range. The results may slightly deviate from former versions of the Swiss Ephemeris, even for epochs inside the time range of the old ephemeris.
3. Until version 1.80, the Swiss Ephemeris ignored the fact that the different JPL ephemerides have a different inherent value of the tidal acceleration of the Moon. Calculations of Delta T must be adjusted to this value in order to get best results for the remote past, especially for ancient observations of the Moon and eclipses. Version 2.0 might result in slightly different values for Delta T when compared with older versions of the Swiss Ephemeris. The correct tidal acceleration is automatically set in the functions swe_set_ephe_path() and swe_set_jpl_file(), depending on the available lunar ephemeris. It can also be set using the function swe_set_tid_acc(). Users who work with different ephemerides at the same time, must be aware of this issue. The default value is that of DE430.

New functionality and improvements:

- Former versions of the Swiss Ephemeris were able to exactly reproduce ephemerides of the Astronomical Almanac. The new version also supports apparent position as given by the JPL Horizons web interface (http://ssd.jpl.nasa.gov/horizons.cgi). Please read the chapter 2.4.5.i in this file above.
- swe_sidtime() was improved so that it give sensible results for the whole time range of DE431.
- swe_time_equ() was improved so that it give sensible results for the whole time range of DE431.
- New functions swe_lmt_to_lat() and swe_lat_to_lmt() were added. They convert local mean time into local apparent time and reverse.
- New function swe_lun_eclipse_when_loc() provides lunar eclipses that are observable at a given geographic position.
- New ayanamsha SE_SID_TRUE_CITRA (= 27, "true chitrapaksha ayanamsha"). The star Spica is always exactly at 180°.
- New ayanamsha SE_SIDM_TRUE_REVATI (= 28), with the star Revati (zeta Piscium) always exactly at 0°.

Bug fixes:

- swetest.c, line 556: geopos[10], array size was too small in former versions
- swetest.c, option -t[time] was buggy
- a minor bugfix in swe_heliacal_ut(): in some cases, the morning last of the Moon was not found if visibility was bad and the geographic latitude was beyond 50N/S.
- unused function swi_str_concat() was removed.

Changes from version 1.79 to 1.80

- Security update: improved some places in code where buffer overflow could occur (thanks to Paul Elliott)
- APC house system
- New function swe_house_name(), returns name of house method
- Two new ayanamshas: Suryasiddhanta Revati (359'50 polar longitude) and Citra (180° polar longitude)
- Bug fix in swehel.c, handling of age of observer (thanks to Victor Reijs).
- Bug fix in swe_lun_occult_when_loc(): correct handling of starting date (thanks to Olivier Beltrami)

Changes from version 1.78 to 1.79

- Improved precision in eclipse calculations: 2^{nd} and 3^{rd} contact with solar eclipses, penumbral and partial phases with lunar eclipses.
- Bug fix in function swe_sol_eclipse_when_loc().If the local maximum eclipse occurs at sunset or sunrise, tret[0] now gives the moment when the lower limb of the Sun touches the horizon. This was not correctly implemented in former versions
- Several changes to C code that had caused compiler warnings (as proposed by Torsten Förtsch).
- Bug fix in Perl functions swe_house() etc. These functions had crashed with a segmentation violation if called with the house parameter 'G'.
- Bug fix in Perl function swe_utc_to_jd(), where gregflag had been read from the 4^{th} instead of the 6^{th} parameter.
- Bug fix in Perl functions to do with date conversion. The default mechanism for gregflag was buggy.

- For Hindu astrologers, some more ayanamshas were added that are related to Suryasiddhanta and Aryabhata and are of historical interest.

Changes from version 1.77 to 1.78

- precession is now calculated according to Vondrák, Capitaine, and Wallace 2011.
- Delta t for current years updated.
- new function: swe_rise_trans_true_hor() for risings and settings at a local horizon with known height.
- functions swe_sol_eclipse_when_loc(), swe_lun_occult_when_loc(): return values tret[5] and tret[6] (sunrise and sunset times) added, which had been 0 so far.
- function swe_lun_eclipse_how(): return values attr[4-6] added (azimuth and apparent and true altitude of moon).
- **Attention** with swe_sol_eclipse_how(): return value attr[4] is azimuth, now measured from south, in agreement with the function swe_azalt() and swe_azalt_rev().
- minor bug fix in swe_rise_trans(): twilight calculation returned invalid times at high geographic latitudes.
- minor bug fix: when calling swe_calc() 1. with SEFLG_MOSEPH, 2. with SEFLG_SWIEPH, 3. again with SEFLG_MOSEPH, the result of 1. and 3. were slightly different. Now they agree.
- minor bug fix in swe_houses(): With house methods H (Horizon), X (Meridian), M (Morinus), and geographic latitudes beyond the polar circle, the ascendant was wrong at times. The ascendant always has to be on the eastern part of the horizon.

Changes from version 1.76 to 1.77

- Delta T:
- Current values were updated.
- File sedeltat.txt understands doubles.
- For the period before 1633, the new formulae by Espenak and Meeus (2006) are used. These formulae were derived from Morrison & Stephenson (2004), as used by the Swiss Ephemeris until version 1.76.02.
- The tidal acceleration of the moon contained in LE405/6 was corrected according to Chapront/Chapront-Touzé/Francou A&A 387 (2002), p. 705.

Fixed stars:

- There was an error in the handling of the proper motion in RA. The values given in fixstars.cat, which are taken from the Simbad database (Hipparcos), are referred to a great circle and include a factor of cos(d0).
- There is a new fixed stars file sefstars.txt. The parameters are now identical to those in the Simbad database, which makes it much easier to add new star data to the file. If the program function swe_fixstar() does not find sefstars.txt, it will try the old fixed stars file fixstars.cat and will handle it correctly.
- Fixed stars data were updated, some errors corrected.
- Search string for a star ignores white spaces.

  Other changes:

- New function swe_utc_time_zone(), converts local time to UTC and UTC to local time. Note, the function has no knowledge about time zones. The Swiss Ephemeris still does not provide the time zone for a given place and time.

- swecl.c: swe_rise_trans() has two new minor features: SE_BIT_FIXED_DISC_SIZE and SE_BIT_DISC_BOTTOM (thanks to Olivier Beltrami)
- minor bug fix in swemmoon.c, Moshier's lunar ephemeris (thanks to Bhanu Pinnamaneni)
- solar and lunar eclipse functions provide additional data:

  attr[8] magnitude, attr[9] saros series number, attr[10] saros series member number

Changes from version 1.75 to 1.76

New features:

- Functions for the calculation of heliacal risings and related phenomena, s. chap. 6.15-6.17.
- Functions for conversion between UTC and JD (TT/UT1), s. chap. 7.2 and 7.3.
- File sedeltat.txt allows the user to update Delta T himself regularly, s. chap. 8.3
- Function swe_rise_trans(): twilight calculations (civil, nautical, and astronomical) added
- Function swe_version() returns version number of Swiss Ephemeris.
- Swiss Ephemeris for Perl programmers using XSUB

Other updates:

- Delta T updated (-2009).

Minor bug fixes:

- swe_house_pos(): minor bug with Alcabitius houses fixed
- swe_sol_eclipse_when_glob(): totality times for eclipses jd2456776 and jd2879654 fixed (tret[4], tret[5])

Changes from version 1.74 to version 1.75

- The Swiss Ephemeris is now able to read ephemeris files of JPL ephemerides DE200 DE421. If JPL will not change the file structure in future releases, the Swiss Ephemeris will be able to read them, as well.
- Function swe_fixstar() (and swe_fixstar_ut()) was made slightly more efficient.
- Function swe_gauquelin_sector() was extended.
- Minor bug fixes.

Changes from version 1.73 to version 1.74

The Swiss Ephemeris is made available under a dual licensing system:

a) GNU public license version 2 or later;

b) Swiss Ephemeris Professional License.

For more details, see at the beginning of this file and at the beginning of every source code file.

Minor bug fixes:

- Bug in swe_fixstars_mag() fixed.
- Bug in swe_nod_aps() fixed. With retrograde asteroids (20461 Dioretsa, 65407 2002RP120), the calculation of perihelion and aphelion was not correct.
- The ephemeris of asteroid 65407 2002RP120 was updated. It had been wrong before 17 June 2008.

Changes from version 1.72 to version 1.73

New features:

- Whole Sign houses implemented (W)
- swe_house_pos() now also handles Alcabitius house method
- function swe_fixstars_mag() provides fixed stars magnitudes

Changes from version 1.71 to version 1.72

- Delta T values for recent years were updated
- Delta T calculation before 1600 was updated to Morrison/Stephenson 2004..
- New function swe_refrac_extended(), in cooperation with archeoastronomer Victor Reijs.

This function allows correct calculation of refraction for altitudes above sea > 0, where the ideal horizon and planets that are visible may have a negative height.

- Minor bugs in swe_lun_occult_when_glob() and swe_lun_eclipse_how() were fixed.

Changes from version 1.70.03 to version 1.71

In September 2006, Pluto was introduced to the minor planet catalogue and given the catalogue number 134340.

The numerical integrator we use to generate minor planet ephemerides would crash with 134340 Pluto, because Pluto is one of those planets whose gravitational perturbations are used for the numerical integration. Instead of fixing the numerical integrator for this special case, we changed the Swiss Ephemeris functions in such a way that they treat minor planet 134340 Pluto (ipl=SE_AST_OFFSET+134340) as our main body Pluto (ipl=SE_PLUTO=9). This also results in a slightly better precision for 134340 Pluto.

Swiss Ephemeris versions prior to 1.71 are not able to do any calculations for minor planet number 134340.

Changes from version 1.70.02 to version 1.70.03

Bug fixed (in swecl.c: swi_bias()): This bug sometimes resulted in a crash, if the DLL was used and the SEFLG_SPEED was not set. It seems that the error happened only with the DLL and did not appear, when the Swiss Ephemeris C code was directly linked to the application.

Code to do with (#define NO_MOSHIER) was removed.

Changes from version 1.70.01 to version 1.70.02

Bug fixed in speed calculation for interpolated lunar apsides. With ephemeris positions close to 0 Aries, speed calculations were completely wrong. E.g. swetest -pc -bj3670817.276275689 (speed = 1448042° !)

Thanks, once more, to Thomas Mack, for testing the software so well.

Changes from version 1.70.00 to version 1.70.01

Bug fixed in speed calculation for interpolated lunar apsides. Bug could result in program crashes if the speed flag was set.

Changes from version 1.67 to version 1.70

Update of algorithms to IAU standard recommendations:

All relevant IAU resolutions up to 2005 have been implemented. These include:

- the "frame bias" rotation from the JPL reference system ICRS to J2000. The correction of position ~= 0.0068 arc sec in right ascension.
- the precession model P03 (Capitaine/Wallace/Chapront 2003). The correction in longitude is smaller than 1 arc second from 1000 B.C. on.
- the nutation model IAU2000B (can be switched to IAU2000A)
- corrections to epsilon
- corrections to sidereal time
- fixed stars input data can be "J2000" or "ICRS"
- fixed stars conversion FK5 -> J2000, where required
- fixed stars data file was updated with newer data
- constants in sweph.h updated

For more info, see the documentation swisseph.doc, chapters 2.1.2.1-3.

New features:

- Ephemerides of "interpolated lunar apogee and perigee", as published by Dieter Koch in 2000 (swetest -pcg).

For more info, see the documentation swisseph.doc, chapter 2.2.4.

- House system according to Bogdan Krusinski (character 'U').

For more info, see the documentation swisseph.doc, chapter 6.1.13.

Bug fixes:

- Calculation of magnitude was wrong with asteroid numbers < 10000 (10-nov-05)

Changes from version 1.66 to version 1.67

Delta-T updated with new measured values for the years 2003 and 2004, and better estimates for 2005 and 2006.

Bug fixed #define SE_NFICT_ELEM 15

(earlier changes no longer documented)

What is missing ?

There are some important limits in regard to what you can expect from an ephemeris module. We do not tell you:

- how to draw a chart;
- which glyphs to use;
- when a planet is stationary;
- how to compute universal time from local time, i.e. what timezone a place is located in;
- how to compute progressions, solar returns, composite charts, transit times and a lot more;
- what the different calendars (Julian, Gregorian ...) mean and when they apply.

Function overview

(incomplete)

Function	Description
swe_azalt	computes the horizontal coordinates (azimuth and altitude) computes either ecliptical or equatorial
swe_azalt_rev	coordinates from azimuth and true altitude
swe_calc	computes the positions of planets, asteroids, lunar nodes and apogees
swe_calc_ut	modified version of swe_calc
swe_close	releases most resources used by the Swiss Ephemeris
swe_cotrans	coordinate transformation, from ecliptic to equator or vice-versa
swe_cotrans_sp	coordinate transformation of position and speed, from ecliptic to equator or vice-versa
date_conversion	time and checks whether a date is legal
swe_degnorm	normalization of any degree number to the range 0 ... 360
swe_deltat	computes the difference between Universal Time (UT, GMT) and Ephemeris time
swe_fixstar	computes fixed stars
swe_fixstar_ut	modified version of swe_fixstar
swe_get_ayanamsa	computes the [ayanamsha
swe_get_ayanamsa_ut	modified version of swe_get_ayanamsa
swe_get_planet_name	finds a planetary or asteroid name by given number
swe_get_tid_acc	gets the tidal acceleration
swe_heliacal_ut	compute heliacal risings etc. of a planet or star
swe_house_pos	compute the house position of a given body for a given ARMC
swe_houses	calculates houses for a given date and geographic position
swe_houses_armc	computes houses from ARMC (e.g. with the composite horoscope which has no date)
swe_houses_ex	the same as swe_houses(). Has a parameter, which can be used, if sidereal house positions are wanted
swe_jdet_to_utc	converts JD (ET/TT) to UTC
swe_jdut1_to_utc	converts JD (UT1) to UTC
swe_julday	conversion from day, month, year, time to Julian date
swe_lat_to_lmt	converts local apparent time (LAT) to local mean time (LMT)
swe_lmt_to_lat	converts local mean time (LMT) to local apparent time (LAT)
swe_lun_eclipse_how	computes the attributes of a lunar eclipse at a given time

Function	Description
swe_lun_eclipse_when	finds the next lunar eclipse
swe_lun_eclipse_when_loc	finds the next lunar eclipse observable from a geographic location
swe_nod_aps	computes planetary nodes and apsides: perihelia, aphelia, second focal points of the orbital ellipses
swe_nod_aps_ut	modified version of swe_nod_aps
swe_pheno	computes phase, phase angle, elongation, apparent diameter, apparent magnitude
swe_pheno_ut	modified version of swe_pheno
swe_refrac	the true/apparent altitude conversion
swe_refrac_extended	the true/apparent altitude conversion
swe_revjul	conversion from Julian date to day, month, year, time
swe_rise_trans	computes the times of rising, setting and meridian transits
swe_rise_trans_true_hor	computes the times of rising, setting and meridian transits relative to true horizon
swe_set_ephe_path	set application's own ephemeris path
swe_set_jpl_file	sets JPL ephemeris directory path
swe_set_sid_mode	specifies the sidereal modes
swe_set_tid_acc	sets tidal acceleration used in swe_deltat()
swe_set_topo	sets what geographic position is to be used before topocentric planet positions for a certain birth place can be computed
swe_sidtime	returns sidereal time on Julian day
swe_sidtime0	returns sidereal time on Julian day, obliquity and nutation
swe_sol_eclipse_how	calculates the solar eclipse attributes for a given geographic position and time
swe_sol_eclipse_when_glob	finds the next solar eclipse globally
swe_sol_eclipse_when_loc	finds the next solar eclipse for a given geographic position
swe_sol_eclipse_where	finds out the geographic position where an eclipse is central or maximal
swe_time_equ	returns the difference between local apparent and local mean time
swe_utc_time_zone	converts UTC int time zone time
swe_version	returns the version of the Swiss Ephemeris
swe_vis_limit_mag	calculates the magnitude for an object to be visible

various functions	Description
swe_csnorm	normalize argument into interval \0..DEG360\
swe_cs2degstr	centiseconds -> degrees string
swe_cs2lonlatstr	centiseconds -> longitude or latitude string
swe_cs2timestr	centiseconds -> time string
swe_csroundsec	round second, but at 29.5959 always down
swe_d2l	double to long with rounding, no overflow check
swe_day_of_week	day of week Monday = 0, ... Sunday = 6
swe_difcs2n	distance in centisecs p1 – p2 normalized to -180..180\
swe_difcsn	distance in centisecs p1 – p2 normalized to \0..360\
swe_difdeg2n	distance in degrees
swe_difdegn	distance in degrees

Appendix

Appendix A

File seorbel.txt with elements of fictitious bodies

```
# Orbital elements of fictitious planets
# 27 Jan. 2000
#
# This file is part of the Swiss Ephemeris, from Version 1.60 on.
#
# Warning! These planets do not exist!
#
# The user can add his or her own elements.
# 960 is the maximum number of fictitious planets.
#
# The elements order is as follows:
# 1. epoch of elements (Julian day)
# 2. equinox (Julian day or "J1900" or "B1950" or "J2000" or "JDATE")
# 3. mean anomaly at epoch
# 4. semi-axis
# 5. eccentricity
# 6. argument of perihelion (ang. distance of perihelion from node)
# 7. ascending node
# 8. inclination
# 9. name of planet
#
# use '#' for comments
# to compute a body with swe_calc(), use planet number
# ipl = SE_FICT_OFFSET_1 + number_of_elements_set,
# e.g. number of Kronos is ipl = 39 + 4 = 43
#
# Witte/Sieggruen planets, refined by James Neely
J1900, J1900, 163.7409, 40.99837, 0.00460, 171.4333, 129.8325, 1.0833, Cupido # 1
J1900, J1900, 27.6496, 50.66744, 0.00245, 148.1796, 161.3339, 1.0500, Hades # 2
J1900, J1900, 165.1232, 59.21436, 0.00120, 299.0440, 0.0000, 0.0000, Zeus # 3
J1900, J1900, 169.0193, 64.81960, 0.00305, 208.8801, 0.0000, 0.0000, Kronos # 4
J1900, J1900, 138.0533, 70.29949, 0.00000, 0.0000, 0.0000, 0.0000, Apollon # 5
J1900, J1900, 351.3350, 73.62765, 0.00000, 0.0000, 0.0000, 0.0000, Admetos # 6
J1900, J1900, 55.8983, 77.25568, 0.00000, 0.0000, 0.0000, 0.0000, Vulcanus # 7
J1900, J1900, 165.5163, 83.66907, 0.00000, 0.0000, 0.0000, 0.0000, Poseidon # 8
#
# Isis-Transpluto; elements from "Die Sterne" 3/1952, p. 70ff.
```

```
# Strubell does not give an equinox. 1945 is taken in order to
# reproduce the as best as ASTRON ephemeris. (This is a strange
# choice, though.)
# The epoch according to Strubell is 1772.76.
# 1772 is a leap year!
# The fraction is counted from 1 Jan. 1772
2368547.66, 2431456.5, 0.0, 77.775, 0.3, 0.7, 0, 0, Isis-Transpluto # 9
# Nibiru, elements from Christian Woeltge, Hannover
1856113.380954, 1856113.380954, 0.0, 234.8921, 0.981092, 103.966, -44.567, 158.708, Nibiru # 10
# Harrington, elements from Astronomical Journal 96(4), Oct. 1988
2374696.5, J2000, 0.0, 101.2, 0.411, 208.5, 275.4, 32.4, Harrington # 11
# according to W.G. Hoyt, "Planets X and Pluto", Tucson 1980, p. 63
2395662.5, 2395662.5, 34.05, 36.15, 0.10761, 284.75, 0, 0, Leverrier (Neptune) # 12
2395662.5, 2395662.5, 24.28, 37.25, 0.12062, 299.11, 0, 0, Adams (Neptune) # 13
2425977.5, 2425977.5, 281, 43.0, 0.202, 204.9, 0, 0, Lowell (Pluto) # 14
2425977.5, 2425977.5, 48.95, 55.1, 0.31, 280.1, 100, 15, Pickering (Pluto) # 15
J1900,JDATE, 252.8987988 + 707550.7341 * T, 0.13744, 0.019, 322.212069+1670.056*T, 47.787931-1670.056*T, 7.
# Selena/White Moon
J2000,JDATE, 242.2205555, 0.05279142865925, 0.0, 0.0, 0.0, 0.0, Selena/White Moon, geo # 17
```

All orbital elements except epoch and equinox may have T terms, where:

T = (tjd – epoch) / 36525.

(See, e.g., Vulcan, the second last elements set (not the "Uranian" Vulcanus but the intramercurian hypothetical planet Vulcan).) "T * T", "T2", "T3" are also allowed.

The equinox can either be entered as a Julian day or as "J1900" or "B1950" or "J2000" or, if the equinox of date is required, as "JDATE". If you use T terms, note that precession has to be taken into account with JDATE, whereas it has to be neglected with fixed equinoxes.

No T term is required with the mean anomaly, i.e. for the speed of the body, because our software can compute it from semi-axis and gravity. However, a mean anomaly T term had to be added with Vulcan because its speed is not in agreement with the laws of physics. In such cases, the software takes the speed given in the elements and does not compute it internally.

The software also accepts orbital elements for fictitious bodies that move about the Earth. As an example, study the last elements set in the excerpt of seorbel.txt above. After the name of the body, ", geo" has to be added. Example: #17 Selena/White Moon