Earth Explorer Mission CFI Software

Quick Start Guide

Code:	EE-MA-DMS-GS-009
Issue:	2.0
Date:	19/01/2009

	Name	Function	Signature
Prepared by:	Juan José Borrego Bote	Project Engineer	
Checked by:	José Antonio González Abeytua	Project Manager	
Approved by:	José Antonio González Abeytua	Project Manager	

DEIMOS Space S.L.
Ronda de Poniente, 19
Edificio Fiteni VI, Portal 2, 2ª Planta
28760 Tres Cantos (Madrid), SPAIN
Tel.: +34 91 806 34 50
Fax: +34 91 806 34 51
E-mail: deimos@deimos-space.com

Document Information

Contract Data		Classification	
Contract Number:	15583/01/NL/GS	Internal	
		Public	
Contract Issuer:	ESA / ESTEC	Industry	X
		Confidential	

External Distribution		
Name	Organisation	Copies

Electronic handling	
Word Processor:	Adobe Framemaker 6.0
Archive Code:	P/SUM/DMS/01/026-056
Electronic file name:	ee-ma-dms-gs-009-10

Document Status Log

Issue	Change Description	Date	Approval
1.0	First Version	09/03/07	
1.1	First Version. Finished document.	21/03/07	
1.2	2nd Version. In line with EXPCFI v3.7	13/07/07	
2.0	In line with EXPCFI v4.0	19/07/09	

Table of Contents

1 ACRONYMS AND NOMENCLATURE

1.1 Acronyms

ANX	Ascending Node Crossing
AOCS	Attitude and Orbit Control Subsystem
ASCII	American Standard Code for Information Interchange
BOM	Beginning Of Mission
CFI	Customer Furnished Item
EOM	End Of Mission
ESA	European Space Agency
ESTEC	European Space Technology and Research Centre
GPL	GNU Public License
GPS	Global Positioning System
IERS	International Earth Rotation Service
I/F	Interface
LS	Leap Second
OBT	On-board Binary Time
OSF	Orbit Scenario File
SRAR	Satellite Relative Actual Reference
SUM	Software User Manual
TAI	International Atomic Time
UTC	Coordinated Universal Time
UT1	Universal Time UT1
WGS[84]	World Geodetic System 1984

2 REFERENCE DOCUMENTS

[MCD]	Earth Explorer Mission CFI Software. Mission Conventions Document. EE-MADMS-GS-0001. Issue 1.4 02/08/07.
[GEN_SUM]	Earth Explorer Mission CFI Software. General Software User Manual. EE-MA-DMS-GS-002. Issue 4.0 19/01/2009.
[F_H_SUM]	Earth Explorer Mission CFI Software. EXPLORER_FILE_HANDLING Software User Manual. EE-MA-DMS-GS-008. Issue 4.0 19/01/2009
[D_H_SUM]	Earth Explorer Mission CFI Software. EXPLORER_DATA_HANDLING Software User Manual. EE-MA-DMS-GS-007. Issue 4.0 19/01/2009
[LIB_SUM]	Earth Explorer Mission CFI Software. EXPLORER_LIB Software User Manual. EE-MA-DMS-GS-003. Issue 4.0 19/01/2009
[ORB_SUM]	Earth Explorer Mission CFI Software. EXPLORER_ORBIT Software User Manual. EE-MA-DMS-GS-004. Issue 4.0 19/01/2009
[PNT_SUM]	Earth Explorer Mission CFI Software. EXPLORER_POINTING Software User Manual. EE-MA-DMS-GS-005. Issue 4.0 19/01/2009
[VIS_SUM]	Earth Explorer Mission CFI Software. EXPLORER_VISIBILITY Software User Manual. EE-MA-DMS-GS-006. Issue 4.0 19/01/2009

3 INTRODUCTION

The Earth Explorer Mission CFI Software is a collection of software functions performing accurate computations of mission related parameters for Earth Explorer missions. The functions are delivered as six software libraries gathering functions that share similar functionalities:

- EXPLORER_FILE_HANDLING: functions for reading and writing files in XML format.
- EXPLORER_DATA_HANDLING: functions for reading and writing Earth Explorer Mission files.
- EXPLORER_LIB: functions for time transformations, coordinate transformations and other basic transformations.
- EXPLORER_ORBIT: functions for computing orbit information.
- EXPLORER_POINTING: functions for pointing calculations.
- EXPLORER_VISIBILITY: functions for getting visibility time segments of the satellite.

A detailed description about the software can be found in the user manuals (see section 2): a general overview and information about how to get and install the software is in [GEN_SUM], while detailed function description appears in the other user manuals, one per library. It is highly recommended to read [GEN_SUM] before going ahead with the current document.

The purpose of the current document is to give complementary information to the user manuals to provide a general view of what the Earth Explorer CFI Software can do and the strategies to follow for the different use cases.

4 EARTH EXPLORER CFI USAGE

The usage cases of the CFI can be classified in the following categories:

- Reading XML files
- Writing XML files
- Reading/writing Earth Explorer Mission files
- Verifying XML files
- Time correlation initialisation
- Time transformations
- Other time calculations
- Using different astronomical models
- Coordinate transformations
- Orbit initialisation
- Orbital calculations
- Orbit propagation
- Orbit interpolation
- Generation of Earth Explorer Mission Orbit Files
- Target calculation:
  - Attitude initialisation.
  - Atmosphere initialisation.
  - DEM.
- Swath calculations
- Visibility calculations
- Time segments manipulation

In the following sections, each case is described together with the strategy to follow to get the desired results. For each case, a set of examples is provided. Besides theses examples, there is a C-program example per library that is distributed with the CFI installation package (see [GEN_SUM] section 6.6)

4.1 CFI Identifiers (Ids)

Before continuing with the usage cases, it is useful to the understand what are the CFI Identifiers (from now on, they will be noted as Ids).

In most cases, CFI functions need to make use of a certain amount of internal data that characterise the system. The way to provide this data to the functions is a variable, the Id. In fact the Id is just a structure that contains all the needed internal data.

Different kinds of Ids have been created to reflect the different categories or "objects" that group the data handled in the CFI. This means that each Id type stores internal data needed for a specific computation. The data stored in the Ids are hidden from the user, however the data can be accessed through a set of specific functions that retrieve the information from the Ids (see the Software User Manuals in section 2).

A list of the Ids used in the CFI is given in the table below:

Table 1: CFI Identifiers

ID	Library	Description	Usage	Dependencies
sat_id	-	Satellite identifier	Input parameter (does not need to be initialised)	Independent, no previous initialisation of any other Id is required
xl_model_id	explorer_lib	It stores the data about the models to be used for astronomical models	Output parameter in the initialisation. Input parameter for model-dependent functions and closing function	Independent, no previous initialisation of any other Id is required. A no-initialised ID can be used
xl_time_id	explorer_lib	It stores the time correlations	Output parameter in the initialisation. Input parameter for time related computations and time closing function	Independent, no previous initialisation of any other Id is required
xo_orbit_id	explorer_orbit	It stores the orbit data needed for orbit calculations	Output parameter in the initialisation. Input parameter in orbit calculations, propagation, interpolation, visibility computations and orbit closing function	xo_orbit_id = sat_id + time_id + (orbit data). It requires that xl_time_id has been previously initialised
xp_atmos_id	explorer_pointing	It stores the atmospheric data used in target functions	Output parameter in the initialisation. Input parameter in target routines and atmospheric closing function	Independent, no previous initialisation of any other Id is required
xp_dem_id	explorer_pointing	It stores the Digital Elevation Model data used in target functions	Output parameter in the initialisation. Input parameter in target routines and DEM closing function	Independent, no previous initialisation of any other Id is required

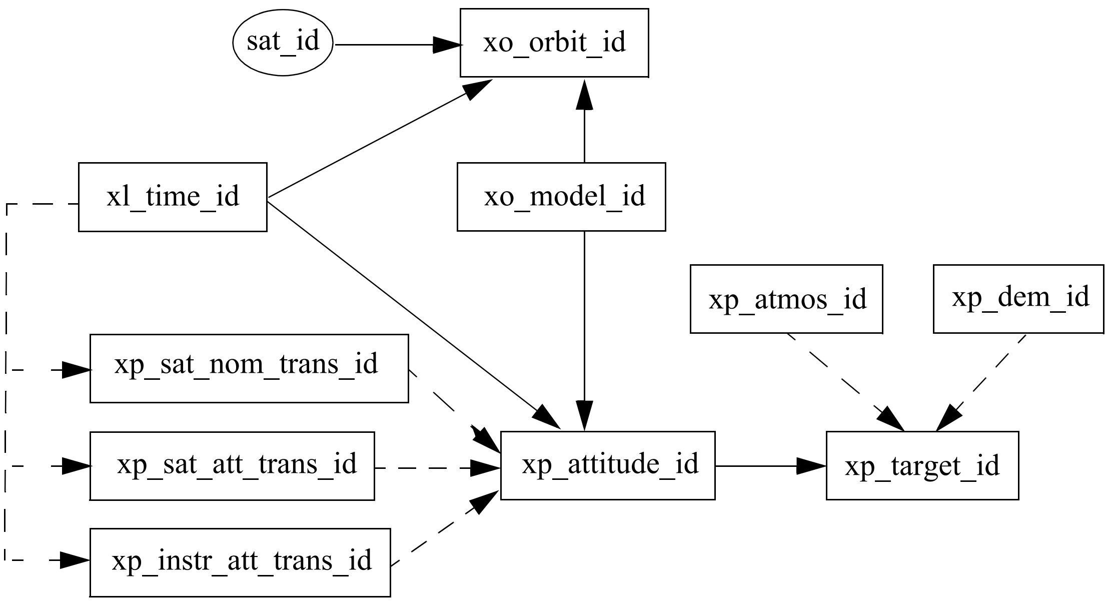
Table 1: CFI Identifiers

xp_sat_nom_trans_id	explorer_pointing	It stores the Satellite Nominal Attitude Ref. Frame data used in attitude functions	Output parameter in the initialisation. Input parameter in attitude routines and satellite nominal attitude transformation closing function	Independent, no previous initialisation of any other Id is required, except when the file initialisation routine is used. Then xp_sat_nom_trans_id requires that xl_time_id has been previously initialised
xp_sat_att_trans_id	explorer_pointing	It stores the Satellite Attitude Ref. Frame data used in attitude functions	Output parameter in the initialisation. Input parameter in attitude routines and satellite attitude transformation closing function	Independent, no previous initialisation of any other Id is required, except when the file initialisation routine is used. Then xp_sat_att_trans_id requires that xl_time_id has been previously initialised
xp_instr_trans_id	explorer_pointing	It stores the Instrument Ref. Frame data used in attitude functions	Output parameter in the initialisation. Input parameter in attitude routines and instrument transformation closing function	Independent, no previous initialisation of any other Id is required, except when the file initialisation routine is used. Then xp_instr_trans_id requires that xl_time_id has been previously initialised
xp_attitude_id	explorer_pointing	It stores the results of the attitude calculation used in target functions	Output parameter in the initialisation. Input parameter in target routines and attitude closing function	xp_attitude_id = xl_time_id + xp_sat_nom_trans_id + xp_sat_att_trans_id + xp_instr_trans_id + attitude computation. It requires that xl_time_id has been previously initialised but it does not necessary require that xp_sat_nom_trans_id, xp_sat_att_trans_id and xp_instr_trans_id have been previously initialised
xp_target_id	explorer_pointing	It stores the results of the target calculation, needed to get ancillary results	Output parameter in the initialisation. Input parameter in extra results target routines and target closing function	xp_target_id = xp_attitude_id + xp_atmos_id + xp_dem_id + target data. It requires that xp_attitude_id has been previously initialised but, it does not necessary require that xp_dem_id and xp_atmos_id _id have been previously initialised
run_id	all	It stores a set of Ids.	It is used for calling functions with simplified interfaces as only the run_id has to be provided	Independent, but all ids that are included in the run_id depend on it, so the run_id has to be freed before the run id.

Note that the last entry in the table is an Id, called *runId,* that includes a group of Ids. All functions that has an Id in the interface, has a equivalent interface that replaces all the Ids for the run_id. This equivalent function has the same name that the original one but ended with the suffix *_run*.

Figure 1 shows the dependency between the Ids.

Figure 1: Hierarchical structure of the initialisation variables in the CFI.



To get a complete description of the Ids, refer to [GEN_SUM].

4.2 Error Handling

A complete description of the error handling for the Earth Explorer CFI functions can be found in [GEN_SUM] section 8.

4.3 Reading XML files

The CFI provides a set of functions for reading XML files, all they within the explorer_file_handling library.

The strategy to read a file is the following:

- Open the file (with **xf_tree_init_parser)**: note that this function returns a number that identifies the file. Every time a file is open, a new number is assigned to the file. The maximum number of XML files that can be opened is 10.
- Read values from the file: The file has to be identified with the number provided by the previous function. There are several ways of reading the file:
  - Sequentially.
  - Random access
- Close the file (with **xf_tree_cleanup_parser**)

A detailed description of the reading process can be found in [F_H_SUM]

Example 4.3 - I: Reading XML files.

Variable declaration

```
    long fd, error;
    char xmlFile[] = "my_xml_file";
    char string_element[] = "First_Tag";
    char string_value[256];
    ...
```

Open file

```
    /* Open file */
    fd = xf_tree_init_parser (xmlFile, &error);
    if ( error < XF_CFI_OK )
    {
      printf("\nError parsing file %s\n", xmlFile);
      return (-1)
    }
```

Reading routines

```
    /* Read the string element value in <First_Tag> */
    xf_tree_read_string_element_value (&fd, string_element, string_value, &error);
    if ( error < XF_CFI_OK )
    {
      printf("\nError reading element as string\n" );
    }
    else
    {
      printf ("Element: %s *** Value: %s\n", string_element, string_value );
    }
```

Close file

```
    /* Close file */
    xf_tree_cleanup_parser (&fd, &error);

    if ( error < XF_CFI_OK )
    {
      printf("\nError freeing file %s\n", xmlFile);
      return(-1);
    }
```

4.4 Writing XML files

The CFI provides a set of functions for writing XML files, all they within the explorer_file_handling library.

The strategy to write a file is the following:

- Create the file (with **xf_tree_create)**: note that this function returns a number that identifies the file. Every time a file is open, a new number is assigned to the file. The maximum number of XML files that can be opened simultaneously is 10.
- Write values in the file: The file has to be identified with the number provided by the previous function.
- Write file to disk (with **xf_tree_write**)
- Close the file (with **xf_tree_cleanup_parser**)

A detailed description of the reading process can be found in [F_H_SUM]

Example 4.4 - I: Writing XML files from scratch.

Variable declaration

```
/* Variables declaration */
long fd, error;
char xmlFile[] = "my_xml_file";
...
```

Create file structure

```
/* Create the file parser */
fd = xf_tree_create (&error);
if ( error < XF_CFI_OK )
{
  printf("\nError parsing file \n");
  return (-1);
}
```

Writing routines

```
/* Create the root element */
xf_tree_create_root (&fd, "Earth_Explorer_File", &error);
if ( error < XF_CFI_OK )
{
  printf("\nError creating file \n");
  return (-1);
}

/* Add a child to the root element */
xf_tree_add_child (&fd, "/Earth_Explorer_File", "First_Tag", &error );
if ( error < XF_CFI_OK )
{
  printf("\nError adding adding a child \n" );
}

/* Add a value to the "First_Tag" */
xf_tree_set_string_node_value ( &fd, ".", "value_1", "%s", &error );
if ( error < XF_CFI_OK )
{
   printf("\nError adding adding a child \n" );
```

```
      }

      /* Add a child to the root element */
      xf_tree_add_next_sibling (&fd, ".", "Second_tag", &error );
      if ( error < XF_CFI_OK )
      {
         printf("\nError adding adding a child \n" );
      }

      xf_tree_set_string_node_value ( &fd, ".", "value_2", "%s", &error );
      if ( error < XF_CFI_OK )
      {
         printf("\nError adding adding a child \n" );
      }
```

Writing routines

```
      /* Write the file to disk */
      xf_tree_write (&fd, xmlFile, &error );
      if ( error < XF_CFI_OK )
      {
        printf("\nWriting Error\n" );
        return(-1);
      }
```

Write file to disk

```
      /* Close file parser */
      xf_tree_cleanup_parser (&fd, &error);
      if ( error < XF_CFI_OK )
      {
        printf("\nError freeing file %s\n", xmlFile);
        return(-1);
      }

      [...]
```

Close file

The resulting file would be as follows:

```
<?xml version="1.0"?>
<Earth_Explorer_File>
  <First_tag>value_1</First_tag>
  <Second_tag>value_2</Second_tag>
</Earth_Explorer_File>
```

4.5 Reading/Writing Earth Explorer files

The Earth Explorer CFI also provides functions for reading and writing the mission files. This way by calling a single function, we can get the content of a file stored in a structure (for the reading case), or we can dump the content of a data structure to a mission file (for the writing case). The following files are supported:

- IERS Bulletin B files
- Orbit files
- Orbit Scenario files
- DORIS Navigator files
- Attitude files
- Star tracker files
- Digital Elevation files (ACE model)
- Swath Definition files
- Swath Template files
- Zone Database files
- Station Database files
- Star Database files

Note that many of the structures used for reading files contain dynamic data that is allocated within the reading function. In these cases, the memory has to be freed when it is not going to be used any more by calling the suitable function.

All this functions are provided in the EXPLORER_DATA_HANDLING library ([D_H_SUM]).

Example 4.5 - I: Reading and writing an Orbit Scenario file

```
/* Variables */

long status, func_id, n;
 long ierr[XD_NUM_ERR_READ_OSF];
char msg[XD_MAX_COD][XD_MAX_STR];
char input_file[] = "OSF_File.EEF"
char output_file[] = "Copy_of_OSF_File.EEF"
xd_osf_file osf_data;
```

Variable declaration

```
/* reading OSF file */
status = xd_read_osf(input_file, &osf_data, ierr);

/* error handling */
if (status != XD_OK)
{
  func_id = XD_READ_OSF_ID;
  xd_get_msg(&func_id, ierr, &n, msg);
  xd_print_msg(&n, msg);
  if (status <= XD_ERR) return(XD_ERR);
}
```

Read File

```
      /* Print results */

      printf("- Number of records      : %ld ", osf_data.num_rec);
      printf("- 1st. Orbital Change: \n");
      printf("     Absolute Orbit: %ld\n", osf_data.osf_rec[0].abs_orb);
      printf("     Cycle days     : %ld\n", osf_data.osf_rec[0].cycle_days);
      printf("     Cycle orbits   : %ld\n", osf_data.osf_rec[0].cycle_orbits);
      printf("     MLST           : %f\n", osf_data.osf_rec[0].mlst);

      [...]
```

Using data structure

```
      /* Writing the OSF file */
      status = xd_write_osf(output_file, &fhr, &osf_data, ierr);

      /* error handling */
      if (status != XD_OK)
      {
        func_id = XD_WRITE_OSF_ID;
        xd_get_msg(&func_id, ierr, &n, msg);
        xd_print_msg(&n, msg);
        if (status <= XD_ERR) return(XD_ERR);
      }
```

Writing another OSF with the same data

```
      [...]
```

```
      /* Free memory */
      xd_free_osf(&osf_data);
```

Free data structure

4.6 Verifying XML files

Most of Earth Explorer files are in XML format. The formats of the files are described in [D_H_SUM]. It is possible to check the format of a file with respect to its XSD schema by calling the function **xd_xml_validate** or using the standalone function **xml_validate**.

Following there are two examples showing the use of this function. For a detailed explanation about the function refer to [D_H_SUM]

Example 4.6 - I: Validating a file with respect to a given schema

Variable declaration & initialisation

```
/* Variables */
Char input_file[256],
     schema[256],
     log_file[256];
long mode, valid_status;

strcpy (input_file, "../data/CRYOSAT_XML_OSF");
mode = XD_USER_SCHEMA;
strcpy(schema, "../../../files/schemas/EO_OPER_MPL_ORBSCT_0100.XSD");
strcpy(logfile, ""); /* => Show the validation outputs in the standard output */
```

File validaton

```
/* Validate the file */
status = xd_xml_validate (input_file, &mode, schema, logfile,
                          &valid_status, ierr);

/* error handling */
if (status != XD_OK)
{
   func_id = XD_XML_VALIDATE_ID;
   xd_get_msg(&func_id, ierr, &n, msg);
   xd_print_msg(&n, msg);
   if (status <= XD_ERR) return(XD_ERR);
}

/* Print output values */
printf("Validation status for %s: [%s]\n", input_file,
        (valid_status == XD_OK)? "VALID" : "INVALID");
```

Example 4.6 - II: Validating a file with respect to the default schema:

```
strcpy(schema, "");
mode = XD_DEFAULT_SCHEMA; /* The schema is taken from the root element
                             in the file*/
/* Validate the file */
status = xd_xml_validate (input_file, &mode, schema, logfile,
                          &valid_status, ierr);
```

```
/* error handling */
if (status != XD_OK)
{
   func_id = XD_XML_VALIDATE_ID;
   xd_get_msg(&func_id, ierr, &n, msg);
   xd_print_msg(&n, msg);
   if (status <= XD_ERR) return(XD_ERR);
}

/* Print output values */
printf("Validation status for %s: [%s]\n", input_file,
        (valid_status == XD_OK)? "VALID" : "INVALID");
```

4.7 Time correlation initialisation

The initialisation of the time correlations does not provide any direct functionality to the user, but it is needed for many other operations within the mission planning.

The initialisation consist on storing the time correlation between the different allowed time references, (i.e. TAI, UTC, UT1 and GPS time) in a *xl_time_id* structure.

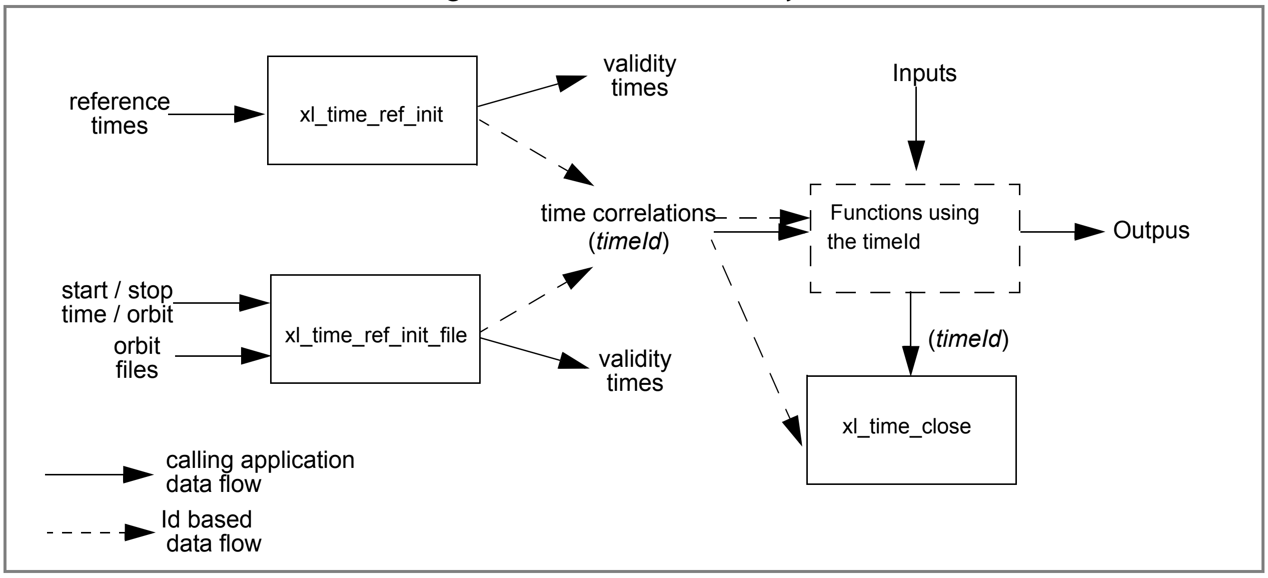
In order to accomplish such correlations, two possible strategies can be used:

- Initialisation from a single or multiple orbit files (**xl_time_ref_init_file**).
- Initialisation from a given set of time references (**xl_time_ref_init**).

After finalising the transformations, the *xl_time_id* must be freed (**xl_time_close**).

Figure 1.represents the data flow for the *xl_time_id* structure.

Figure 1: xl_time_id data flow



Examples showing the usage of the time initialisation can be found in section 4.8.

4.8 Time transformations

The Earth Explorer CFI Software contains a set of functions to transform an input time in a given time reference and format to another time reference and/or format.

Time transformations functions requires the user to initialise the time correlations if the time reference is going to be changed[1](see section 4.7). Once the initialisation has been performed, the user is able to transform any date expressed in one of the allowed time references to another, through the Time Format / Reference Transformation functions. The *xl_time_id* has to be provided to each of these functions. The process can be repeated as needed without initialising the time correlations each time.

For a complete description of all the time transformation function refer to [LIB_SUM].

Besides the time transformation functions, there exists a program called **time_conv** that performs the same calculation (see Example 4.8 - III)

Example 4.8 - I: Time transformations. Initialisation with an IERS file.

```
/* Variables */
long   status, func_id, n;
long   xl_ierr[XL_ERR_VECTOR_MAX_LENGTH];
char   msg[XL_MAX_COD][XL_MAX_STR];

xl_time_id time_id = {NULL};

long   time_model, n_files, time_init_mode, time_ref;
char   *time_file[2];
double time0, time1, val_time0, val_time1;
long   orbit0, orbit1;
long   ierr[XL_NUM_ERR_TIME_REF_INIT_FILE];
char   iers_file[] = "../data/bulb.dat";

long   format_in, ref_in,
       format_out, ref_out;
long   transport_in[4];
char   ascii_in[XD_MAX_STR], ascii_out[XD_MAX_STR];
double proc_out;
```

Variable declaration

```
/* Time initialisation */

time_model     = XL_TIMEMOD_IERS_B_PREDICTED;
n_files        = 1;
time_init_mode = XL_SEL_TIME;
time_ref       = XL_TIME_TAI;
time0          = 240.0;
time1          = 260.0;
orbit0         = 0; /* dummy */
orbit1         = 0; /* dummy */
time_file[0]   = iers_file;
```

Time Initialisation

1. When the output time reference is equal to the input one, there is no need of initialiasing the time_id

```
    status = xl_time_ref_init_file (&time_model, &n_files, time_file,
                                    &time_init_mode, &time_ref, &time0, &time1,
                                    &orbit0, &orbit1, &val_time0, &val_time1,
                                    &time_id, xl_ierr);
    /* error handling */
    if (status != XL_OK)
    {
       func_id = XL_TIME_REF_INIT_FILE_ID;
       xl_get_msg(&func_id, xl_ierr, &n, msg);
       xl_print_msg(&n, msg);
       if (status <= XL_ERR) return(XL_ERR);
    }
```

Time Initialisation

```
    /* 1st. Time transformation: time in TAI and standard transport format to
       GPS time in standard ASCII format */
    format_in  = XL_TRANS_STD;
    ref_in     = XL_TIME_TAI;
    format_out = XL_ASCII_STD_REF_MICROSEC;
    ref_out    = XL_TIME_GPS;

    transport_in[0] = 245;    /* TAI time [integer days]         */
    transport_in[1] = 150;    /* TAI time [integer seconds]      */
    transport_in[2] = 1500;   /* TAI time [integer microseconds] */
    transport_in[3] = 0;      /* Unused in Transport_Standard    */

    status = xl_time_transport_to_ascii(&time_id,
                                        &format_in,  &ref_in,  transport_in,
                                        &format_out, &ref_out, ascii_out,
                                        xl_ierr);
    /* error handling */
    if (status != XL_OK)
    {
       func_id = XL_TIME_TRANSPORT_TO_ASCII_ID;
       xl_get_msg(&func_id, t2a_ierr, &n, msg);
       xl_print_msg(&n, msg);
       if (status <= XL_ERR) return(XL_ERR);
    }

    /* Print input/output values */
    printf("- Transport input format: %ld \n", format_in);
    printf("- Input time_reference  : %ld \n"  , ref_in);
    printf("- Input transport time  : %ld, %ld, %ld \n",
            transport_in[0], transport_in[1], transport_in[2]);
    printf("- ASCII input format    : %ld \n", format_out);
    printf("- Output time reference : %ld \n", ref_out);
    printf("- Output ASCII time     : %s \n", ascii_out);
```

Time Operations

```
/* 2nd. Time transformation: time in GPS and standard ASCII format to
   processing format and UT1 time reference */
format_in  = format_out;
ref_in     = ref_out;
format_out = XL_PROC;
ref_out    = XL_TIME_UT1;
strcpy(ascci_in, ascii_out);

status = xl_time_ascii_to_processing(&time_id,
                                     &format_in,  &ref_in,  ascii_in,
                                     &format_out, &ref_out, proc_out,
                                     xl_ierr);
/* error handling */
if (status != XL_OK)
{
   func_id = XL_TIME_ASCII_TO_PROCESSING_ID;
   xl_get_msg(&func_id, t2a_ierr, &n, msg);
   xl_print_msg(&n, msg);
   if (status <= XL_ERR) return(XL_ERR);
}

[...]
```

Time Operations

```
/* Close time references */
status = xl_time_close(&time_id, xl_ierr);
if (status != XL_OK)
{
   func_id = XL_TIME_CLOSE_ID;
   xl_get_msg(&func_id, xl_ierr, &n, msg);
   xl_print_msg(&n, msg);
   if (status <= XL_ERR) return(XL_ERR);
}
```

Close Time Correlations

Example 4.8 - II: Time transformations. Initialisation with given time correlations.

```
/* Variables */

long   status, func_id, n;
long   xl_ierr[XL_ERR_VECTOR_MAX_LENGTH];
char   msg[XL_MAX_COD][XL_MAX_STR];
double tri_time[4];
double tri_orbit_num, tri_anx_time, tri_orbit_duration;
xl_time_id time_id = {NULL};

long format_in, format_out,
     ref_in, ref_out;
double proc_in;
```

Variable declaration

```
/* Time initialisation */
tri_time[0] = -245.100000000;            /* TAI time [days] */
tri_time[1] = tri_time[0] - 35.0/86400.; /* UTC time [days] (= TAI - 35.0 s) */
tri_time[2] = tri_time[0] - 35.3/86400.; /* UT1 time [days] (= TAI - 35.3 s) */
tri_time[3] = tri_time[0] - 19.0/86400.; /* GPS time [days] (= TAI - 19.0 s) */

tri_orbit_num = 10;
tri_anx_time  = 5245.123456;
tri_orbit_duration = 6035.928144;

status = xl_time_ref_init(tri_time, &tri_orbit_num, &tri_anx_time,
                          &tri_orbit_duration, &time_id, tri_ierr);
/* error handling */
if (status != XL_OK)
{
   func_id = XL_TIME_REF_INIT_ID;
   xl_get_msg(&func_id, xl_ierr, &n, msg);
   xl_print_msg(&n, msg);
   if (status <= XL_ERR) return(XL_ERR);
}
```

Time Initialisation

```
/* time from TAI to UT1 time reference in processing format */
format_in  = XL_PROC;
ref_in     = XL_TIME_TAI;
format_out = XL_PROC;
ref_out    = XL_TIME_UT1;
proc_in    = 0.0;

status = xl_time_processing_to_processing(&time_id,
                                          &format_in,  &ref_in,  proc_in,
                                          &format_out, &ref_out, proc_out,
                                          xl_ierr);
/* error handling */
if (status != XL_OK)
{
   func_id = XL_TIME_PROCESSING_TO_PROCESSING_ID;
   xl_get_msg(&func_id, t2a_ierr, &n, msg);
   xl_print_msg(&n, msg);
   if (status <= XL_ERR) return(XL_ERR);
}

[...]
```

Time Operations

```
/* Close time references */
status = xl_time_close(&time_id, xl_ierr);
if (status != XL_OK)
{
   func_id = XL_TIME_CLOSE_ID;
   xl_get_msg(&func_id, xl_ierr, &n, msg);
   xl_print_msg(&n, msg);
   if (status <= XL_ERR) return(XL_ERR);
}
```

Example 4.8 - III: Time transfromation with executable file.

The following command line does the same transformation than the code in Example 4.8 - III:

```
time_conv -fmt_in PROC -fmt_out PROC -ref_in TAI -ref_out UT1 -day 0.0 -v
          -tai 0.0000 -gps 0.00021991 -utc 0.00040509 -ut1 0.00040865
```

4.9 Other time calculations

Besides the time transformation functions shown in section 4.8, the CFI provide functions for:

- Operation between Dates
  - **xl_time_add**: adds a duration to a TAI, UTC, UT1 or GPS time expressed in Processing format.
  - **xl_time_diff**: subtracts two TAI, UTC, UT1 or GPS times expressed in Processing format.
- Transformations from/to On-board Times
  - • **xl_time_obt_to_time**: transforms an On-board Time (OBT) into a TAI, UTC, UT1 or GPS time in processing format.
  - **xl_time_time_to_obt**: transforms a TAI, UTC, UT1 or GPS time expressed in Processing format into an On-board Time (OBT).

These functions do not need to follow any special strategy and can be called from any part of the program without having to initialise the timeId.

Example 4.9 - I: Adding two dates.

```
/* Variables */
long   status, func_id, n;
long   xl_ierr[XL_ERR_VECTOR_MAX_LENGTH];
char   msg[XL_MAX_COD][XL_MAX_STR];

double proc_1, proc_2, proc_out;
long   proc_id, time_ref;

proc_id  = XL_PROC;
time_ref = XL_TIME_TAI;
proc_1 = 245.100001;   /* Processing Time, MJD2000 [days] */
proc_2 = 110.123456;   /* Added duration [days] */

/* Call xl_time_add function */
status = xl_time_add(&proc_id, &time_ref, &proc_1, &proc_2,
                     &proc_out, xl_ierr);

/* Error handling */
if (status != XL_OK)
{
   func_id = XL_TIME_ADD_ID;
   xl_get_msg(&func_id, tad_ierr, &n, msg);
   xl_print_msg(&n, msg);
   if (status <= XL_ERR) return(XL_ERR);
}

/* Print output values */
printf("- Output time (TAI) = %12.12lf + %12.12lf = %12.12lf days",
        proc_1, proc_2, proc_out);
[...]
```

4.10 Using different astronomical models

The EXPCFI software allows the user to choose the models for the Earth shape and astronomical calculations. The models that can be chosen are grouped in the following categories (for further details refer to [LIB_SUM]):

- Earth
- Sun
- Moon
- Planet
- Star
- Nutation
- Precession
- Physical and astronomical constants

In order to select the models with which the EXPCFI has to work, a CFI ID called model_id has been created (see [GEN_SUM], section 7.3). The calling sequence for a C program where the model_id is needed, would be as follows:

- Declare the model_id variable:

  xl_model_id model_id = {NULL};

  The model_id has to be initialised this way (as other CFI ID's), so that the EXPCFI could recognise that the model_id is not initialised.
- Optionally, initialise the model_id with **xl_model_init**. This function would set the requested models in the model_id. If the model_id is not initialised, the EXPCFI functions will use the default models.
- The model_id is used as an input parameter in the EXPCFI functions if it is needed.
- Close the model_id with **xl_model_close** (Only if the model_id was initialised).

This strategy can be seen in the **Example 4.11 - I.** For other examples, the default models will be used (mode_id no initialised).

4.11 Coordinate transformations

The Earth Explorer CFI software provides a set of functionality for coordinate transformations:

- Transformations between reference frames: It is possible to transform between the following reference frames: Galactic, Heliocentric, Barycentric Mean of 1950, Barycentric Mean of 2000, Geocentric Mean of 2000, Mean of Date, True of Date, Earth Fixed, Topocentric.

  This transformations are carried out by the following functions: **xl_change_cart_cs**, **xl_topocentric_to_ef** and **xl_ef_to_topocentric**.
- Transformations between Euler's angles and its equivalent rotation matrix (**xl_euler_to_matrix** and **xl_matrix_to_euler**)
- Rotate vectors and compute the rotation angles between two orthonormal frames (**xl_get_rotated_vectors** and **xl_get_rotation_angles**).
- Transformations between vectors and quaternions (**xl_quaternions_to_vectors** and **xl_vectors_to_quaternions**)
- Coordinate Transformations between Geodetic and Cartesian coordinates (**xl_geod_to_cart** and **xl_cart_to_geod**)
- Transformations between cartesian coordinates right ascension and declination angles (**xl_cart_to_radec** and **xl_radec_to_cart**)
- Transformations between Keplerian elements and Cartesian coordinates (**xl_kepl_to_cart** and **xl_cart_to_kepl**)
- Calculation of the osculating true latitude for a cartesian state vector (**xl_position_on_orbit)**

All the functions are described in [LIB_SUM]

xl_change_cart_cs and **xl_position_on_orbit**, require the time initialisation before they are called, so the strategy to follow is the same as for the time transformations functions (see section 4.7 to know more about how to initialise the time correlations). The other functions do not need any special action before calling them.

Example 4.11 - I: Coordinate transformation

```
/* Variables */
long   status, func_id, n;
long   xl_ierr[XL_ERR_VECTOR_MAX_LENGTH];
char   msg[XL_MAX_COD][XL_MAX_STR];

xl_time_id  time_id   = {NULL};
xl_model_id model_id  = {NULL};

long   model_mode,
       models[XL_NUM_MODEL_TYPES_ENUM];
long   cs_in, cs_out;
long   calc_mode = XL_CALC_POS_VEL_ACC;
long   time_ref  = XL_TIME_TAI;
double time-2456.0;

double pos[3] = {-6313910.323647, 3388282.485785, 0.002000};
double vel[3] = {531.059763, 971.331224, 7377.224410};
double acc[3] = {-0.175235, 0.095468, 0.000000};
```

Variable declaration

```
    [... Time initialisation... ]

/* Model initialisation:
   models set to default models.
   "models" array does not need to be initialised */
model_mode = XL_MODEL_DEFAULT;
status = xl_model_init(&model_mode, models,
                        &model_id, xl_ierr);
if (status != XL_OK)
{
   func_id = XL_MODEL_INIT_ID;
   xl_get_msg(&func_id, xl_ierr, &n, msg);
   xl_print_msg(&n, msg);
   if (status <= XL_ERR) return(XL_ERR);
}
```

Model initialisation

```
cs_in  = XL_TOD;    /* Initial coordinate system = True of Date */
cs_out = XL_EF;     /* Final coordinate system = Earth fixed */
ext_status = xl_change_cart_cs(&model_id, &time_id,
                               &calc_mode, &cs_in, &cs_out,
                               &time_ref, &time_t, pos, vel, acc,
                               &calc_mode, &cs_in, &cs_out,
                               pos_out, vel_out, acc_out);
if (ext_status != XL_OK)
{
   func_id = XL_CHANGE_CART_CS_ID;
   xl_get_msg(&func_id, &ext_status, &n, msg);
   xl_print_msg(&n, msg);
   if (ext_status <= XL_ERR) return(XL_ERR);
}
/* Print output values */
printf("EF Position     : %lf, %lf, %lf\n",
        pos_out[0], pos_out[1], pos_out[2]);
printf("EF Velocity     : %lf, %lf, %lf\n",
        vel_out[0], vel_out[1], vel_out[2]);
printf("EF Acceleration: %lf, %lf, %lf\n",
        acc_out[0], acc_out[1], acc_out[2]);
[...]
```

Change coordinate system

```
/* Transform to geodetic coordinates */
ext_status = xl_cart_to_geod(&model_id,
                             &calc_mode, pos_out, vel_out,
                             &lon, &lat, &h, &lond, &latd, &hd);
if (ext_status != XL_OK)
{
   func_id = XL_CART_TO_GEOD_ID;
   xl_get_msg(&func_id, &ext_status, &n, msg);
   xl_print_msg(&n, msg);
   if (ext_status <= XL_ERR) return(XL_ERR);
}

/* Print output values */
printf("- Geocentric longitude [deg]        : %lf ", lon_t);
printf("- Geodetic latitude [deg]           : %lf ", lat_t);
printf("- Geodetic altitude [m]             : %lf ", h_t);
printf("- Geocentric longitude rate [deg/s] : %lf ", lond_t);
printf("- Geodetic latitude rate [deg/s]    : %lf ", latd_t);
printf("- Geodetic altitude rate [m/s]      : %lf ", hd_t);
```

Get geodetic coordinates

```
/* Close model initialisation */
status = xl_model_close(&model_id, xl_ierr);
if (status != XL_OK)
{
   func_id = XL_MODEL_CLOSE_ID;
   xl_get_msg(&func_id, xl_ierr, &n, msg);
   xl_print_msg(&n, msg);
   if (status <= XL_ERR) return(XL_ERR);
}
```

Close Model ID

```
[... Close Time initialisation... ]
```

4.12 Orbit initialisation

In order to get orbit related information it is needed to provide some data about the orbit. These data have to be stored in the *xo_orbit_id* (see section 4.1) before any other calculation involving orbital data could be done. These calculations where the *xo_orbit_id* structure are needed are:

- Transformations between time and orbit number
- Getting orbit information
- Orbit propagatin and interpolation

The strategy to follow for initialising the orbit and the afterward usage can be summarize in the following steps:

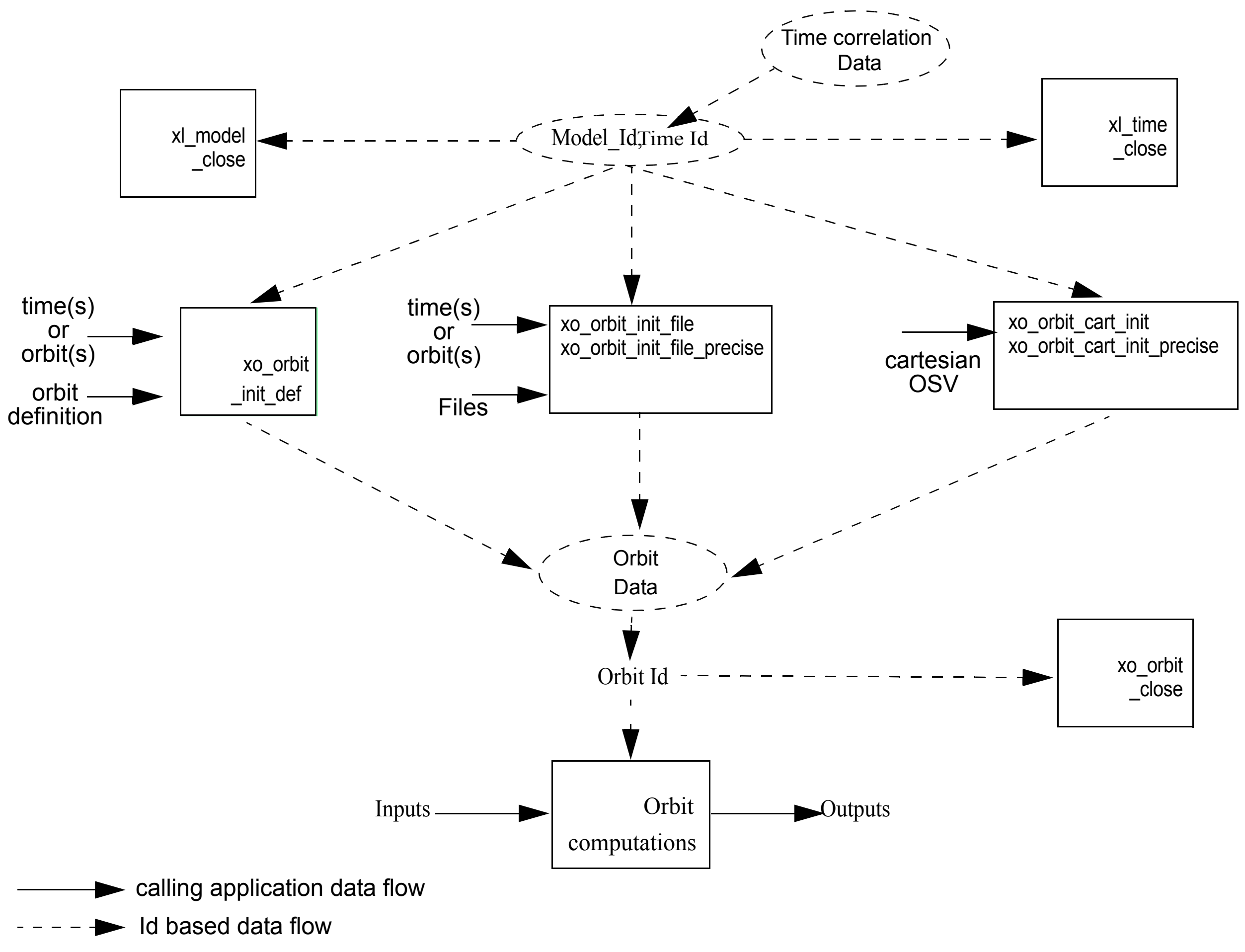
- Time correlation initialisation (see section 4.7): the *xl_time_id* is needed for the orbital initialisation in the next step.
- Orbital initialisation (getting the *xo_orbit_id*): In this step, the user provides orbital information that will be used in further calculations. The data are stored in the *xo_orbit_id* “object”. There are three ways of initialising the orbit:
  - Providing information about the orbital geometry with **xo_orbit_init_def**.
  - Providing a osculating state vector for a given time and orbit number (see function **xo_orbit_cart_init**).
  - Providing orbit files through the function **xo_orbit_init_file**: The orbital files usually contain time correlation data. To ensure that orbit routines produce correct results, these same time correlations should be in the orbit file and the *xl_time_id.*
- Orbit computations: getting orbital information, propagation, interpolation.
- Close orbital initialisation by calling **xo_orbit_close**.
- Close Time initialisation.

A whole description of the functions can be found in [ORB_SUM].

The Figure 2 shows the data flow for the orbital calculations.

The sections 4.13 and 4.14 contain examples showing the orbit initialization usage.

Figure 2: Orbit Information Routines Data Flow



4.13 Orbit calculations

The Earth Explorer CFI functions allow to get the following orbital information for a satellite:

- Transformation between time and orbits: It is possible to know the orbit number and the time after the ANX for a given input time and viceversa (functions **xo_time_to_orbit** and **xo_orbit_to_time**)
- Orbital parameters and orbital numbers (functions **xo_orbit_info**, **xo_orbit_rel_from_abs**, **xo_orbit_abs_from_rel**, **xo_orbit_abs_from_phase**)
- times for which an input set of Sun zenit angles are reached, Sun ocultations by the Earth and Sun ocultations by the Moon (function **xv_orbit_extra**). See Example 4.13 - II.

A whole description of the functions can be found in [ORB_SUM] and [VIS_SUM].

All this functions require the orbit initialisation (section 4.12). The *xo_orbit_id* can be computed with whatever initialisation function, except for the functions that compute the orbit numbers (**xo_orbit_rel_from_abs**, **xo_orbit_abs_from_rel**, **xo_orbit_abs_from_phase**), for which the *xo_orbit_id* has to be initialised with **xo_orbit_init_file** using an Orbit Scenario file.

Example 4.13 - I: Orbital calculations with xo_orbit_init_def

```
/* Variables */
long   status, func_id, n;
char   msg[XL_MAX_COD][XL_MAX_STR];
long   xl_ierr[XL_ERR_VECTOR_MAX_LENGTH];
long   xo_ierr[XO_ERR_VECTOR_MAX_LENGTH];

long         sat_id     = XO_SAT_CRYOSAT;
xl_time_id   time_id    = {NULL};
xl_model_id  model_id   = {NULL};
xo_orbit_id  orbit_id   = {NULL};

double tri_time[4];
double tri_orbit_num, tri_anx_time, tri_orbit_duration;

long irep, icyc, iorb0, iorb;
double ascmlst, rlong, ascmlst_drift, inclination;
double time0, time;

long abs_orbit, rel_orbit, cycle, phase;
double result_vector[XO_ORBIT_INFO_EXTRA_NUM_ELEMENTS];

long orbit_t, second_t, microsec_t;
long time_ref = XL_TIME_UTC;
double time_t;
```

Variable declaration

```
    /* Time initialisation */
    tri_time[0] = -245.100000000;            /* TAI time [days] */
    tri_time[1] = tri_time[0] - 35.0/86400.; /* UTC time [days] (= TAI - 35.0 s) */
    tri_time[2] = tri_time[0] - 35.3/86400.; /* UT1 time [days] (= TAI - 35.3 s) */
    tri_time[3] = tri_time[0] - 19.0/86400.; /* GPS time [days] (= TAI - 19.0 s) */

    tri_orbit_num = 10;
    tri_anx_time  = 5245.123456;
    tri_orbit_duration = 6035.928144;

    status = xl_time_ref_init(tri_time, &tri_orbit_num, &tri_anx_time,
                              &tri_orbit_duration, &time_id, xl_ierr);
    /* error handling */
    if (status != XL_OK)
    {
       func_id = XL_TIME_REF_INIT_ID;
       xl_get_msg(&func_id, xl_ierr, &n, msg);
       xl_print_msg(&n, msg);
       if (status <= XL_ERR) return(XL_ERR);     /* CAREFUL: normal status */
    }
```

Time initialisation

```
/* Orbit initialisation: xo_orbit_init_def */
irep    = 369;          /* Repeat cycle of the reference orbit [days] */
icyc    = 5344;         /* Cycle length of the reference orbit [orbits] */
ascmlst = 8.6667;       /* Mean local solar time at ANX [hours] */
rlong   = -36.2788;     /* Geocentric longitude of the ANX [deg] */
iorb0   = 0;            /* Absolute orbit number of the reference orbit */
ascmlst_drift = -179.208556;
inclination   = 0.0;

time_init_mode = XO_SEL_ORBIT;
drift_mode     = XO_NOSUNSYNC_DRIFT;
time0 = -2456.0;        /* UTC time in MJD2000 (1993-04-11  00:00:00) [days] */
time =  0.0;             /* Dummy */

/* Calling to xo_orbit_init_def */
status = xo_orbit_init_def(&sat_id, &model_id, &time_id,
                           &time_ref, &time0, &iorb0,
                           &drift_mode, &ascmlst_drift, &inclination,
                           &irep, &icyc, &rlong, &ascmlst,
                           &val_time0, &val_time1, &orbit_id, xo_ierr);
/* error handling */
if (status != XO_OK)
{
   func_id = XO_ORBIT_INIT_DEF_ID;
   xo_get_msg(&func_id, xo_ierr, &n, msg);
   xo_print_msg(&n, msg);
}
```

Orbit initialisation

```
/* Get orbit info */
abs_orbit = 100;
status = xo_orbit_info (&orbit_id, &abs_orbit, result_vector, xo_ierr);

/* error handlinng */
if (status != XO_OK)
{
         func_id = XO_ORBIT_INFO_ID;
         xo_get_msg(&func_id, xo_ierr, &n, msg);
         xo_print_msg(&n, msg);
}

/* print results */
printf("\n\t- Absolute orbit = %ld", abs_orbit);
printf("\n\t- Repeat cycle = %lf", result_vector[0]);
printf("\n\t- Cycle length = %lf", result_vector[1]);
[...]

/* Get time for a given Orbit and ANX time */
orbit_t     = 1034;
second_t    = 3000;
microsec_t = 50;

status = xo_orbit_to_time(&orbit_id,
                          &orbit_t,  &second_t, &microsec_t,
                          &time_ref, &time_t, xo_ierr);
/* error handlinng */
if (status != XO_OK)
{
   func_id = XO_ORBIT_TO_TIME_ID;
   xo_get_msg(&func_id, xo_ierr, &n, msg);
   xo_print_msg(&n, msg);
}

/* Get the Orbit and ANX time from the input time*/
status=xo_time_to_orbit(&orbit_id, &time_ref, &time_t,
                        &orbit_t, &second_t, &microsec_t, xo_ierr);
/* error handlinng */
if (status != XO_OK)
{
   func_id = XO_TIME_TO_ORBIT_ID;
   xo_get_msg(&func_id, xo_ierr, &n, msg);
   xo_print_msg(&n, msg);
}
```

```
/* Close orbit initialisation */
status = xo_orbit_close(&orbit_id, xo_ierr);
if (status != XO_OK)
{
   func_id = XO_ORBIT_CLOSE_ID;
   xo_get_msg(&func_id, xo_ierr, &n, msg);
   xo_print_msg(&n, msg);
}
```

Orbit Close

```
/* Close time references */
status = xl_time_close(&time_id, xl_ierr);
if (status != XL_OK)
{
   func_id = XL_TIME_CLOSE_ID;
   xl_get_msg(&func_id, xl_ierr, &n, msg);
   xl_print_msg(&n, msg);
   if (status <= XL_ERR) return(XL_ERR);
}
```

Time Close

```
   [...]
```

Example 4.13 - II: Orbital calculations with xo_orbit_init_file

```
/* Variables */
long   status, func_id, n;
char   msg[XL_MAX_COD][XL_MAX_STR];
long   xl_ierr[XL_ERR_VECTOR_MAX_LENGTH];
long   xo_ierr[XO_ERR_VECTOR_MAX_LENGTH];

long         sat_id    = XO_SAT_CRYOSAT;
xl_time_id   time_id   = {NULL};
xl_model_id  model_id  = {NULL};
xo_orbit_id  orbit_id  = {NULL};

double tri_time[4];
double tri_orbit_num, tri_anx_time, tri_orbit_duration;

long n_files, time_mode, orbit_mode, time_ref;
char orbit_scenario_file[XD_MAX_STR];
char *files[2];

long abs_orbit, rel_orbit, cycle, phase;
double result_vector[XO_ORBIT_INFO_EXTRA_NUM_ELEMENTS];
long num_sza;
double sza, sza_up, sza_down,
       eclipse_entry, eclipse_exit,
       sun_moon_entry, sun_moon_exit;
```

Variable declaration

Time initialisation

```
/* Time initialisation */
tri_time[0] = -245.100000000;                /* TAI time [days] */
tri_time[1] = tri_time[0] - 35.0/86400.; /* UTC time [days] (= TAI - 35.0 s) */
tri_time[2] = tri_time[0] - 35.3/86400.; /* UT1 time [days] (= TAI - 35.3 s) */
tri_time[3] = tri_time[0] - 19.0/86400.; /* GPS time [days] (= TAI - 19.0 s) */

tri_orbit_num = 10;
tri_anx_time  = 5245.123456;
tri_orbit_duration = 6035.928144;

status = xl_time_ref_init(tri_time, &tri_orbit_num, &tri_anx_time,
                          &tri_orbit_duration, &time_id, xl_ierr);
/* error handling */
if (status != XL_OK)
{
   func_id = XL_TIME_REF_INIT_ID;
   xl_get_msg(&func_id, xl_ierr, &n, msg);
   xl_print_msg(&n, msg);
   if (status <= XL_ERR) return(XL_ERR);
}
```

Orbit initialisation

```
/* Orbit initialisation: xo_orbit_init_file */
n_files = 1;
time_mode = XO_SEL_FILE;
orbit_mode = XO_ORBIT_INIT_OSF_MODE;
time_ref = XO_TIME_UT1;
strcpy(orbit_scenario_file, "../data/CRYOSAT_XML_OSF");
files[0] = orbit_scenario_file;

status = xo_orbit_init_file(&sat_id, &model_id, &time_id,
                            &orbit_mode, &n_files, files,
                            &time_mode, &time_ref,
                            &time0, &time1, &orbit0, &orbit1,
                            &val_time0, &val_time1,
                            &orbit_id, xo_ierr);
/* error handling */
if (status != XO_OK)
{
   func_id = XO_ORBIT_INIT_FILE_ID;
   xo_get_msg(&func_id, xo_ierr, &n, msg);
   xo_print_msg(&n, msg);
}
```

```
/* Get orbit info */
abs_orbit = 100;
status = xo_orbit_info (&orbit_id, &abs_orbit, result_vector, xo_ierr);
if (status != XO_OK)
{
   func_id = XO_ORBIT_INFO_ID;
   xo_get_msg(&func_id, xo_ierr, &n, msg);
   xo_print_msg(&n, msg);
}

/* Get orbit extra info:
   Note that this function uses as input the result_vector from
   xo_orbit_info */
num_sza = 2;
sza[0] = 90;
sza[1] = 80;
status = xo_orbit_extra (&orbit_id, &abs_orbit, result_vector,
                         &num_sza, sza, &sza_up, &sza_down,
                         &eclipse_entry, &eclipse_exit,
                         &sun_moon_entry, &sun_moon_exit,
                         xv_ierr);
if (status != XO_OK)
{
   func_id = XV_ORBIT_EXTRA_ID;
   xv_get_msg(&func_id, xv_ierr, &n, msg);
   xv_print_msg(&n, msg);
}

/* Get relative orbit number and phase */
status = xo_orbit_rel_from_abs (&orbit_id, &abs_orbit,
                                &rel_orbit, &cycle, &phase, xo_ierr);
/* error handlinng */
if (status != XO_OK)
{
   func_id = XO_ORBIT_REL_FROM_ABS_ID;
   xo_get_msg(&func_id, xo_ierr, &n, msg);
   xo_print_msg(&n, msg);
}
```

Orbit functions

```
/* Close orbit_id*/
status = xo_orbit_close(&orbit_id, xo_ierr);
[...]
```

Orbit Close

```
/* Close time references */
status = xl_time_close(&time_id, xl_ierr);
[...]
```

Time Close

4.14 State vector computation (Propagation/Interpolation)

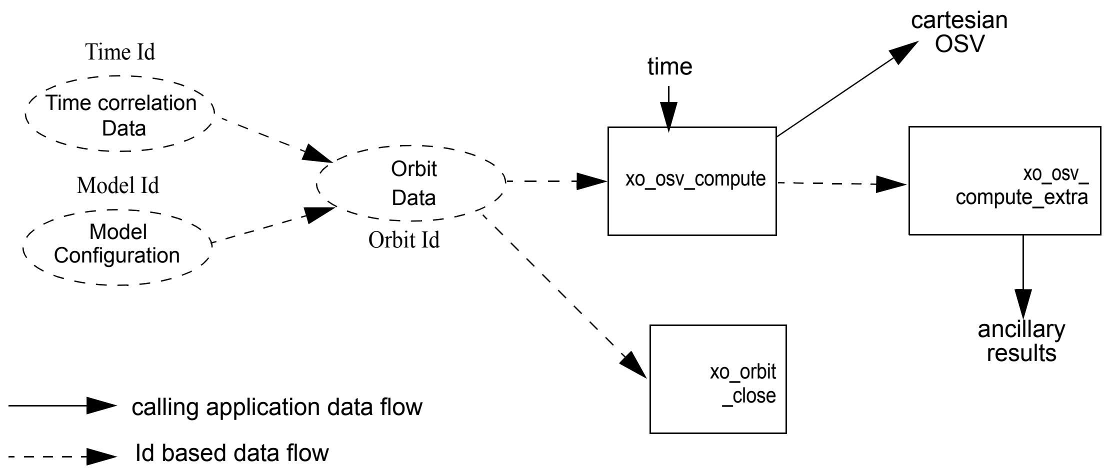
The object of this functionality is the accurate prediction of osculating Cartesian state vectors for user requested times. It is also possible to get ancillary results such as mean and osculating Keplerian orbit state vectors, satellite osculating true latitude, latitude rate and latitude rate-rate, Sun zenith angle and many more.

The propagation/interpol strategy is the following:

- Initialise the time correlations (section 4.7)
- Orbit initialisation (section 4.12) with any of the initialization routines for orbit: **xo_orbit_init_def**, **xo_orbit_init_file** or **xo_orbit_cart_init**.
- Compute the orbital state vector for the required time by calling the function **xo_osv_compute**. The input time has to be within the validity times for the computations. These validity times can be get with the function **xo_orbit_get_osv_compute_valitdiy**.
- Optionally, to obtain ancillary results the user might call the **xo_osv_compute_extra** function.

The following figure shows the data flow for the computation of state vectors:

Figure 3: Propag Routines Data Flow



All the previous function are described in [ORB_SUM].

Example 4.14 - I: Orbit computation

```
/* Variables */
long   status, func_id, n;
char   msg[XL_MAX_COD][XL_MAX_STR];
long   xl_ierr[XL_ERR_VECTOR_MAX_LENGTH];
long   xo_ierr[XO_ERR_VECTOR_MAX_LENGTH];

long          sat_id     = XO_SAT_CRYOSAT;
xl_time_id    time_id    = {NULL};
xl_model_id   model_id   = {NULL};
xo_orbit_id   orbit_id   = {NULL};

double tri_time[4];
double tri_orbit_num, tri_anx_time, tri_orbit_duration;

long time_ref;
double time;
double pos_ini[3], vel_ini[3],
       pos[3],     vel[3];
xo_validity_time val_times;
double val_time0, val_time1;
long abs_orbit;
```

Variable declaration

```
/* Time initialisation */

tri_time[0] = -245.100000000;             /* TAI time [days] */
tri_time[1] = tri_time[0] - 35.0/86400.; /* UTC time [days] (= TAI - 35.0 s) */
tri_time[2] = tri_time[0] - 35.3/86400.; /* UT1 time [days] (= TAI - 35.3 s) */
tri_time[3] = tri_time[0] - 19.0/86400.; /* GPS time [days] (= TAI - 19.0 s) */

tri_orbit_num = 10;
tri_anx_time  = 5245.123456;
tri_orbit_duration = 6035.928144;

status = xl_time_ref_init(tri_time, &tri_orbit_num, &tri_anx_time,
                          &tri_orbit_duration, &time_id, xl_ierr);
if (status != XL_OK)
{
   func_id = XL_TIME_REF_INIT_ID;
   xl_get_msg(&func_id, xl_ierr, &n, msg);
   xl_print_msg(&n, msg);
   if (status <= XL_ERR) return(XL_ERR);
}
```

Time initialisation

```
/* Orbit initialisation */
time_ref = XL_TIME_UT1;
time = -2452.569;
pos_ini[0] = 6427293.5314;
pos_ini[1] = -3019463.3246;
pos_ini[2] = 0;

vel_ini[0] = -681.1285;
vel_ini[1] = -1449.8649;
vel_ini[2] = 7419.5081;

status = xo_orbit_cart_init(&sat_id, &model_id, &time_id,
                            &time_ref, &time,
                            pos_ini, vel_ini, &abs_orbit,
                            &val_time0, &val_time1, &orbit_id,
                            xo_ierr);
if (status != XO_OK)
{
   func_id = XO_ORBIT_CART_INIT_ID;
   xo_get_msg(&func_id, ierr, &n, msg);
   xo_print_msg(&n, msg);
}
```

Orbit initialisation

```
/* Get propagation validity interval*/
init_mode = XO_SEL_DEFAULT; /* select the default time */

status = xo_orbit_get_osv_compute_validity(&orbit_id, &val_times);
if (status == XO_OK)
{
    printf("\t- Propagation validity times = ( %lf , %lf )\n",
                 val_times.start, val_times.stop );
}
```

Get Propagation Validity Times

```
/* propagation: loop to propagate along the validity interval */
time_ref = val_times.time_ref;
for ( time = val_times.start;
      time < val_times.stop;
      time += ((val_time1-val_time0)/10) )
{
   status = xo_osv_compute(&orbit_id, &propag_model, &time_ref, &time,
                           pos, vel, acc, xo_ierr);
   if (status != XO_OK)
   {
      func_id = XO_OSV_COMPUTE_ID;
      xo_get_msg(&func_id, ierr, &n, msg);
      xo_print_msg(&n, msg);
   }

   printf("\t- Time         = %lf\n", time );
   printf("\t- Position     = (%lf, %lf, %lf)\n", pos[0], pos[1], pos[2]);
   printf("\t- Velocity     = (%lf, %lf, %lf)\n", vel[0], vel[1], vel[2]);
   printf("\t- Acceleration = (%lf, %lf, %lf)\n", acc[0], acc[1], acc[2]);
}
```

Orbit State Vector computation

```
/* Close orbit_id */
status = xo_orbit_close(&orbit_id, xo_ierr);
if (status != XO_OK)
{
   func_id = XO_ORBIT_CLOSE_ID;
   xo_get_msg(&func_id, xo_ierr, &n, msg);
   xo_print_msg(&n, msg);
}
```

Orbit close

```
/* close time reference */
status = xl_time_close(&time_id, xl_ierr);
if (status != XO_OK)
{
   func_id = XL_TIME_CLOSE_ID;
   xo_get_msg(&func_id, xl_ierr, &n, msg);
   xo_print_msg(&n, msg);
}
```

Time close

4.15 Generation of Earth Explorer Orbit Mission Files

The Earth Explorer files allow the generation of different orbit files types:

- Orbit Scenario files: **xo_gen_osf_create**.
- Predicted Orbit files: **xo_gen_pof**
- Restituted Orbit files (DORIS restituted and DORIS precise): **xo_gen_rof**
- DORIS Navigator files: **xo_gen_dnf**
- Orbit Event files: **xo_gen_oef**.

The strategy to follow in all cases is similar:

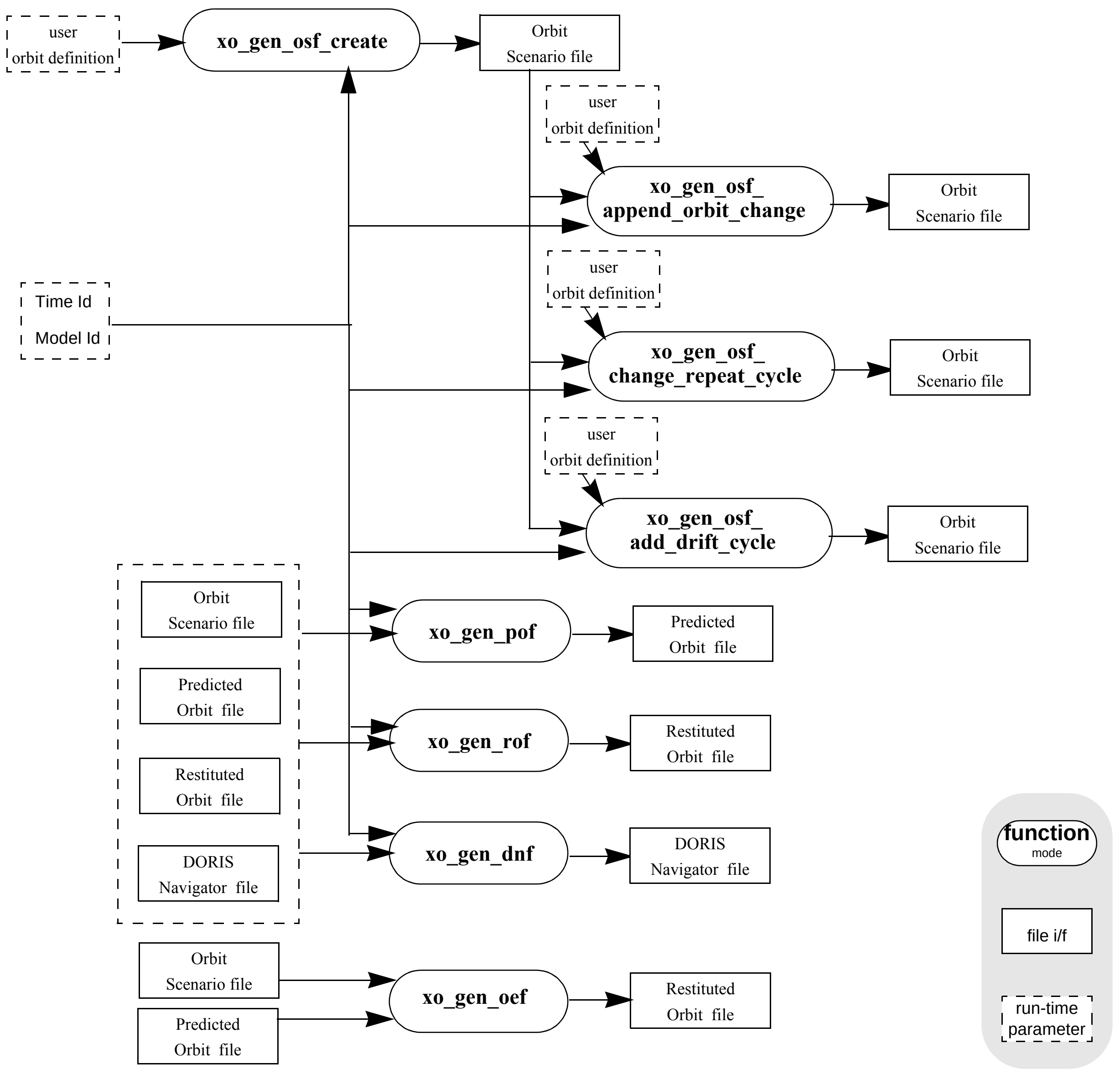
- Initialise the time correlations (see section 4.7) to create the *xl_time_id* that will be used in the generation functions. This step is not needed for the generation of orbit event files.
- Call one of the generation function described above.
- Optionally for the generation of orbit scenario files: it is possible to add orbital changes within the orbit scenario file by calling one of this functions: **xo_gen_osf_append_orbit_change**, **xo_gen_osf_repeath_cycle**, **xo_gen_osf_add_drift_cycle**.
- Close time correlations (see section 4.7). This step is not needed for the generation of orbit event files.

Additionnally there exists a set of executable programs that are equivalent to the previous functions.

More information can be found in [ORB_SUM].

Figure 4 shows the calling sequence for the file generation functions.

Figure 4: File Generation Calling Sequence



Example 4.15 - I: Orbit Scenario file generation

Variable declaration

```
/* Variables */
long   status, func_id, n;
char   msg[XL_MAX_COD][XL_MAX_STR];
long   xl_ierr[XL_ERR_VECTOR_MAX_LENGTH];
long   xo_ierr[XO_ERR_VECTOR_MAX_LENGTH];

long         sat_id          = XO_SAT_CRYOSAT;
xl_time_id   time_id         = {NULL};
xl_model_id  model_id        = {NULL};

double tri_time[4];
double tri_orbit_num, tri_anx_time, tri_orbit_duration;

long   abs_orbit,cycle, phase, phase_inc;
long   repeat_cycle,cycle_length;
long   drift_mode;
double mlst_drift,mlst,anx_long;
long   osf_version = 1;
char   file_class[] = "TEST";
char   fh_system = "CFI Example";

char   output_dir[]    = "";
char   output_file_1[] = "my_osf.eef"           /* name for the output osf */
char   output_file_2[] = "osf_after_append.eef" /* name for the output osf */
```

Time Initialisation

```
/* Time initialisation */

tri_time[0] = -245.100000000;              /* TAI time [days] */
tri_time[1] = tri_time[0] - 35.0/86400.; /* UTC time [days] (= TAI - 35.0 s) */
tri_time[2] = tri_time[0] - 35.3/86400.; /* UT1 time [days] (= TAI - 35.3 s) */
tri_time[3] = tri_time[0] - 19.0/86400.; /* GPS time [days]  (= TAI - 19.0 s)  */

tri_orbit_num = 10;
tri_anx_time   = 5245.123456;
tri_orbit_duration = 6035.928144;

status = xl_time_ref_init(tri_time, &tri_orbit_num, &tri_anx_time,
                          &tri_orbit_duration, &time_id, xl_ierr);
if (status != XL_OK)
{
   func_id = XL_TIME_REF_INIT_ID;
   xl_get_msg(&func_id, xl_ierr, &n, msg);
   xl_print_msg(&n, msg);
   if (status <= XL_ERR) return(XL_ERR);
}
```

```
/* Generate the OSF */

date = 1643.39513888889; /* UTC=2004-07-01_09:29:00.000000 */
abs_orbit = 1;
cycle = 1;
phase = 1;
repeat_cycle = 369;
cycle_length = 5344;
drift_mode = XO_NOSUNSYNC_DRIFT;
mlst_drift = -179.208556;
mlst = 12.0;
anx_long = 37.684960;
osf_version = 1;

status = xo_gen_osf_create(&sat_id, &model_id, &time_id, &abs_orbit,
                           &cycle, &phase, &repeat_cycle, &cycle_length,
                           &anx_long, &drift_mode, &inclination,
                           &mlst_drift, &mlst, &date,
                           output_dir, output_file,
                           file_class, &osf_version, fh_system,
                           xo_ierr);
if (status != XO_OK)
{
    func_id = XO_GEN_OSF_CREATE_ID;
    xo_get_msg(&func_id, xo_ierr, &n, msg);
    xo_print_msg(&n, msg);
}
```

```
/*  Append an orbital change to the generated OSF */

old_nodal_period = 86400.0*(1+mlst_drift/86400.0)*
                   (double)repeat_cycle/(double)cycle_length;
osf_version++;
abs_orbit = 5345;
phase_inc = XO_NO_PHASE_INCREMENT;
repeat_cycle = 369;
cycle_length = 5344;
/* small change wrt to nominal to check tolerances */
anx_long = 37.68497;
mlst = mlst + mlst_drift*(5345-1)*old_nodal_period/(3600.0*86400.0) + 24.0;

status = xo_gen_osf_append_orbit_change(&sat_id, &model_id, &time_id,
                                        output_file_1, &abs_orbit,
                                        &repeat_cycle, &cycle_length,
                                        &anx_long,&drift_mode,
                                        &inclination, &mlst_drift,
                                        &mlst, &phase_inc,
                                        output_dir, output_file_2,
                                        file_class, &osf_version, fh_system,
                                        xo_ierr);
if (status != XO_OK)
{
   func_id = XO_GEN_OSF_APPEND_ORBIT_CHANGE_ID;
   xo_get_msg(&func_id, ierr, &n, msg);
   xo_print_msg(&n, msg);
}
```

Append an orbital change

```
    /* Close time references */
    status = xl_time_close(&time_id, xl_ierr);
    [...]
```

Time Close

Example 4.15 - II: Predicted Orbit file generation

Variable declaration

```
/* Variables */
long   status, func_id, n;
char   msg[XL_MAX_COD][XL_MAX_STR];
long   xl_ierr[XL_ERR_VECTOR_MAX_LENGTH];
long   xo_ierr[XO_ERR_VECTOR_MAX_LENGTH];

long        sat_id   = XO_SAT_CRYOSAT;
xl_model_id model_id = {NULL};
xl_time_id  time_id  = {NULL};

double tri_time[4];
double tri_orbit_num, tri_anx_time, tri_orbit_duration;

char reference_file[] = "input_osf_file";
char pof_filename[XD_MAX_STRING]     = "";
char output_directory[XD_MAX_STRING] = "";

long time_mode,time_ref;
double start_time, stop_time;
double osv_location;
long ref_filetype;

char   file_class[] = "TEST";
long   version_number = 1;
char   fh_system = "CFI Example";
```

Time initialisation

```
/* Time initialisation */
tri_time[0] = -245.100000000;           /* TAI time [days] */
tri_time[1] = tri_time[0] - 35.0/86400.; /* UTC time [days] (= TAI - 35.0 s) */
tri_time[2] = tri_time[0] - 35.3/86400.; /* UT1 time [days] (= TAI - 35.3 s) */
tri_time[3] = tri_time[0] - 19.0/86400.; /* GPS time [days] (= TAI - 19.0 s) */

tri_orbit_num = 10;
tri_anx_time  = 5245.123456;
tri_orbit_duration = 6035.928144;

status = xl_time_ref_init(tri_time, &tri_orbit_num, &tri_anx_time,
                          &tri_orbit_duration, &time_id, xl_ierr);
if (status != XL_OK)
{
   func_id = XL_TIME_REF_INIT_ID;
   xl_get_msg(&func_id, xl_ierr, &n, msg);
   xl_print_msg(&n, msg);
   if (status <= XL_ERR) return(XL_ERR);
}
```

```
/* Generate the POF */
time_mode = XO_SEL_TIME;
time_ref = XO_TIME_UTC;
start_time = 1646.0;
stop_time  = 1647.0;
osv_location = 0.0;
ref_filetype = XO_REF_FILETYPE_OSF;

status = xo_gen_pof(&sat_id, &model_id, &time_id,
                    &time_mode, &time_ref, &start_time,
                    &stop_time, &start_orbit, &stop_orbit,
                    &osv_location, &ref_filetype,
                    reference_file, &pof_filetype, output_directory,
                    pof_filename, file_class, &version_number, fh_system,
                    xo_ierr);
if (status != XO_OK)
{
   func_id = XO_GEN_POF_ID;
   xo_get_msg(&func_id, xo_ierr, &n, msg);
   xo_print_msg(&n, msg);
}
```

Generate Predicted Orbit file

```
/* Close time references */
status = xl_time_close(&time_id, xl_ierr);
[...]
```

Time Close

Example 4.15 - III: Restituted Orbit file generation

```
/* Variables */
long    status, func_id, n;
char    msg[XL_MAX_COD][XL_MAX_STR];
long    xl_ierr[XL_ERR_VECTOR_MAX_LENGTH];
long    xo_ierr[XO_ERR_VECTOR_MAX_LENGTH];
long         sat_id           = XO_SAT_CRYOSAT;
xl_model_id  model_id         = {NULL};
xl_time_id   time_id          = {NULL};
double tri_time[4], tri_orbit_num, tri_anx_time, tri_orbit_duration;
char reference_file[] = "input_osf_file";
char rof_filename[XD_MAX_STRING]      = "";
char output_directory[XD_MAX_STRING] = "";
long time_mode,time_ref;
double start_time, stop_time, osv_interval;
long ref_filetype, osv_precise, rof_filetype;
char    file_class[] = "TEST";
long    version_number = 1;
char    fh_system = "CFI Example";
```

Variable declaration

```
    /* Time initialisation */
    tri_time[0] = -245.100000000;            /* TAI time [days] */
    tri_time[1] = tri_time[0] - 35.0/86400.; /* UTC time [days] (= TAI - 35.0 s) */
    tri_time[2] = tri_time[0] - 35.3/86400.; /* UT1 time [days] (= TAI - 35.3 s) */
    tri_time[3] = tri_time[0] - 19.0/86400.; /* GPS time [days] (= TAI - 19.0 s) */
    tri_orbit_num = 10;
    tri_anx_time  = 5245.123456;
    tri_orbit_duration = 6035.928144;

    status = xl_time_ref_init(tri_time, &tri_orbit_num, &tri_anx_time,
                              &tri_orbit_duration, &time_id, xl_ierr);
    if (status != XL_OK)
    {
       func_id = XL_TIME_REF_INIT_ID;
       xl_get_msg(&func_id, xl_ierr, &n, msg);
       xl_print_msg(&n, msg);
       if (status <= XL_ERR) return(XL_ERR);
   }
```

Time initialisation

```
    /* Generate the ROF */
    time_mode = XO_SEL_TIME;
    time_ref = XO_TIME_UTC;
    start_time = 1646.0;
    stop_time  = 1646.2;
    osv_interval = 60;
    osv_precise  = XO_OSV_PRECISE_MINUTE;
    ref_filetype = XO_REF_FILETYPE_OSF;
    rof_filetype = XO_REF_FILETYPE_ROF;

    status = xo_gen_rof(&sat_id, &model_id, &time_id,
                        &time_mode, &time_ref, &start_time,
                        &stop_time, &start_orbit, &stop_orbit,
                        &osv_interval, &osv_precise, &ref_filetype,
                        reference_file, &rof_filetype, output_directory,
                        rof_filename, file_class, &version_number, fh_system,
                        xo_ierr);
    if (status != XO_OK)
    {
       func_id = XO_GEN_ROF_ID;
       xo_get_msg(&func_id, xo_ierr, &n, msg);
       xo_print_msg(&n, msg);
    }
```

Generate ROF

```
        /* Close time references */
        status = xl_time_close(&time_id, xl_ierr);
        [...]
```

Time Close

Example 4.15 - IV: Executable program for generating a Restituted orbit file:

The following command line generates tha same file that the code in Example 4.15 - III

```
gen_rof -sat CRYOSAT -tref UTC -tstart 1646.0 -tstop 1646.2 -osvint 60     \
        -reftyp OSF -ref input_osf_file                                     \
        -roftyp ROF -rof ROF_example_file.EEF                               \
        -tai 0.0000 -gps 0.00021991 -utc 0.00040509 -ut1 0.00040865
```

Example 4.15 - V: Orbit Event file generation

```
    /* Variables */
    long   status, func_id, n;
    char   msg[XL_MAX_COD][XL_MAX_STR];
    long   xo_ierr[XO_ERR_VECTOR_MAX_LENGTH];

    char   file_class[] = "TEST";
    long   version_number = 1;
    char   fh_system = "CFI Example";

    char oef_filename[XD_MAX_STR];
    char osf_filename[] = "input_osf.eef";
    char pof_filename[] = "input_pof.eef";
```

Variable declaration

```
    /* Generate the OEF */
    status = xo_gen_oef(oef_filename, osf_filename, pof_filename,
                        file_class, &version_number, fh_system,
                        xo_ierr);
    if (status != XO_OK)
    {
       func_id = XO_GEN_OEF_ID;
       xo_get_msg(&func_id, xo_ierr, &n, msg);
       xo_print_msg(&n, msg);
    }
```

OEF generation

4.16 Target calculation

This functionality allows to perform accurate computation of pointing parameters from and to a satellite for various types of targets.

Before the user could call targets function, some parameters has to be initialised:

- Attitude: The attitude defines the relation between coordinate frames related to the satellite and a general reference frame. In order to define the attitude, the user has to call some initialisation functions that generate another CFI Id called *xp_attitude_id*. (See section 4.16.1 for further details about attitude initialisation)
- For some targets calculation it could be needed to take into account the atmospheric refraction of a signal travelling to/from the satellite. In these cases the user could choose the atmospheric model to use. For using an atmospheric model in the target calculation, a CFI Id called *xp_atmos_id* has to be initialised previously, afterwards it is introduced in the target functions.(See section 4.16.3 for further details about atmospheric initialisation)
- For geolocation routines it could be needed a digital elevation model (DEM) in order to provide a more accurate target. The DEM is introduced in the target calculation using the CFI Id structured called *xp_dem_id*. This Id has to be initialised previously to the target calculation.(See section 4.16.4 for further details about DEM initialisation)

4.16.1 Attitude initialisation

The initialisation strategy for the attitude is the following:

- Satellite and instrument attitude frames initialisation. There are three different levels of attitude frames defined for this issue (see [MCD]):
  - Satellite Nominal Attitude Frame.
  - Satellite Attitude Frame
  - Instrument Attitude Frame

  Each of the frames is defined independently and produce a CFI Id where the initialisation parameters are stored. Note that not all attitude frames has to be defined. There are a set of functions to initialise each frame depending on the type of parameters used to establish the reference frame (see Figure 5, Figure 6 and Figure 7)
- Attitude initialisation. Using the function **xp_attitude_init**, the CFI Id *xp_attitude_id* is initialised. At this stage, the structure doesn’t contain attitude data and it cannot be used in target functions.
- Attitude computation: Using a satellite state vector at a given time and the attitude frames previously initialised, the *xp_attitude_id* structure is filled in. by calling the function **xp_attitude_compute.**

All functions for attitude computation are explained in detail in [PNT_SUM].

The typical data flow for the attitude functions described above is shown schematically in the Figure 8.

Figure 5: Satellite Nominal Initialisation

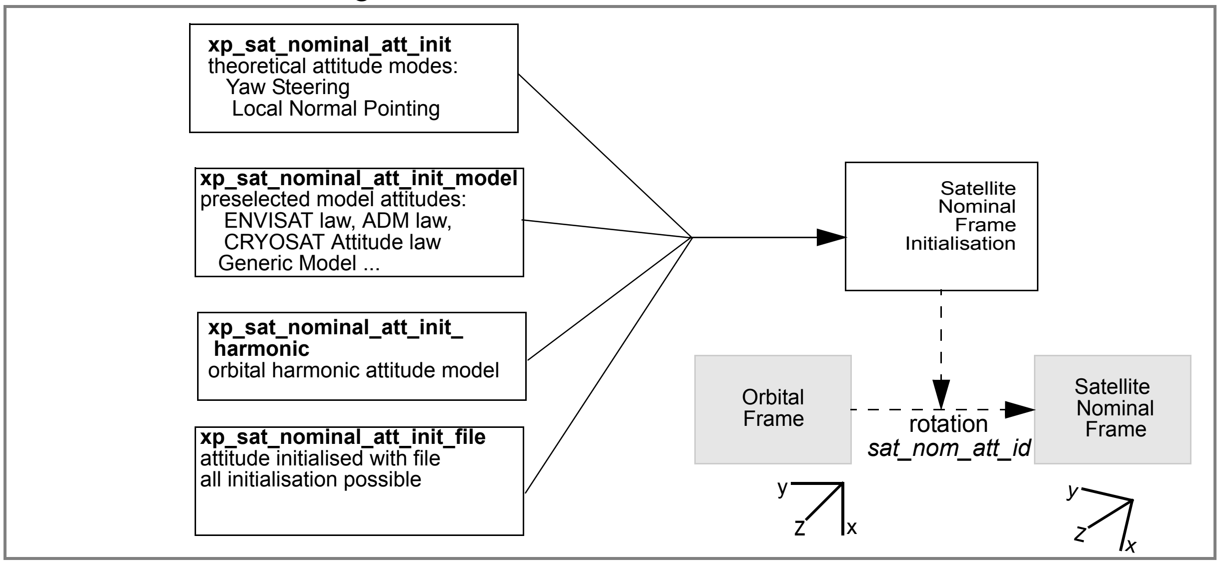

xp_sat_nominal_att_init
theoretical attitude modes:
Yaw Steering
Local Normal Pointing
xp_sat_nominal_att_init_model
preselected model attitudes:
ENVISAT law, ADM law,
CRYOSAT Attitude law
Generic Model ...
xp_sat_nominal_att_init_
harmonic
orbital harmonic attitude model
xp_sat_nominal_att_init_file
attitude initialised with file
all initialisation possible
Satellite Nominal Frame Initialisation
Orbital Frame
rotation
sat_nom_att_id
Satellite Nominal Frame
y z x
y z x

Figure 6: Satellite Initialisation

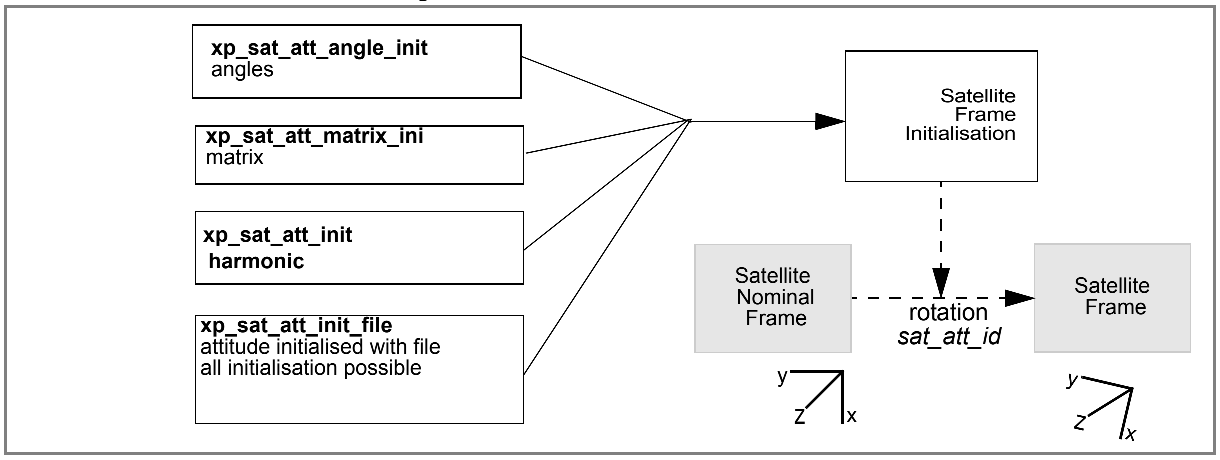


Figure 7: Instrument Initialisation

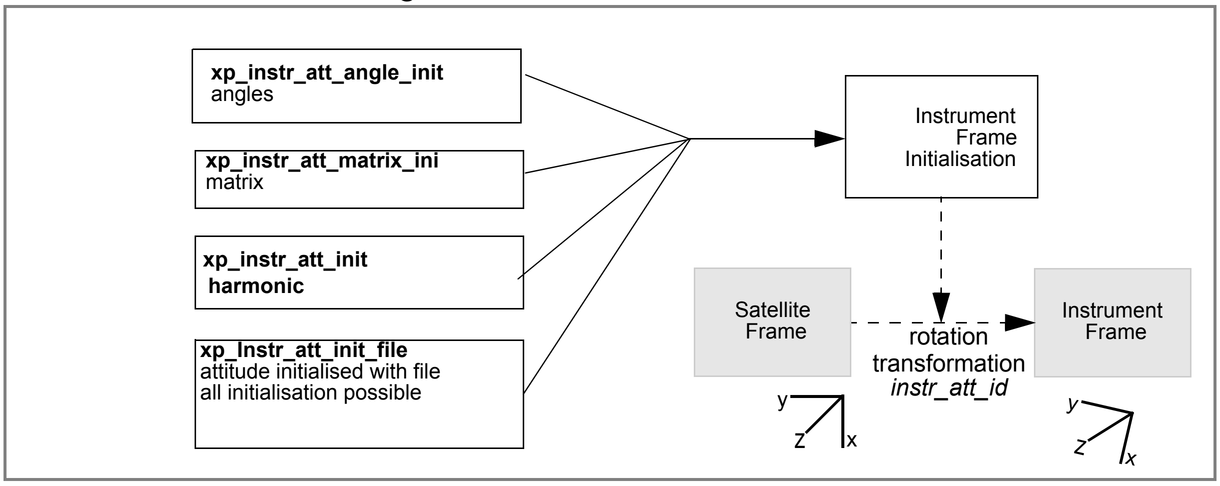
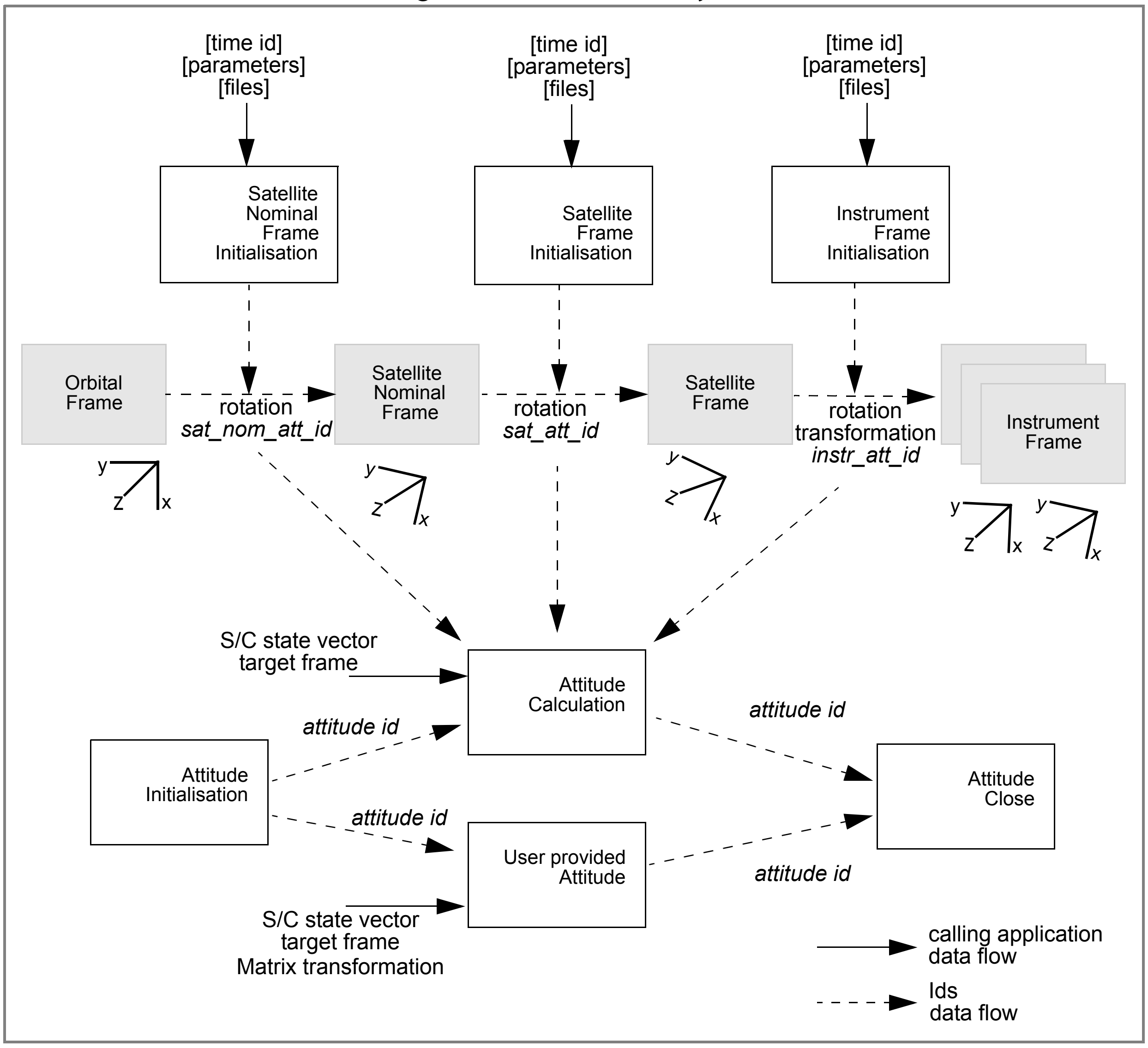


Figure 8: Attitude data flow



Example 4.16 - I: ENVISAT AOCS model plus mispointing angles.

```
/* Variables */
long   status, func_id, n;
char   msg[XL_MAX_COD][XL_MAX_STR];
long   xl_ierr[XL_ERR_VECTOR_MAX_LENGTH];
long   xp_ierr[XP_ERR_VECTOR_MAX_LENGTH];

long                 sat_id           = XO_SAT_ENVISAT;
xl_model_id          model_id         = {NULL};
xl_time_id           time_id          = {NULL};
xp_sat_nom_trans_id  sat_nom_trans_id = {NULL};
xp_sat_trans_id      sat_trans_id     = {NULL};
xp_instr_trans_id    instr_trans_id   = {NULL};
xp_attitude_id       attitude_id      = {NULL};

double tri_time[4];
double tri_orbit_num, tri_anx_time, tri_orbit_duration;

long model_enum;
double model_param[XP_NUM_MODEL_PARAM];
double ang[3];
xp_param_model_str param_model;

long time_ref;
double time;
double pos[3], vel[3], acc[3];
```

Variable declaration

```
/* Time initialisation */
tri_time[0]  = -245.100000000;            /* TAI time [days] */
tri_time[1] = tri_time[0] - 35.0/86400.; /* UTC time [days] (= TAI - 35.0 s) */
tri_time[2] = tri_time[0] - 35.3/86400.; /* UT1 time [days] (= TAI - 35.3 s) */
tri_time[3] = tri_time[0] - 19.0/86400.; /* GPS time [days] (= TAI - 19.0 s) */

tri_orbit_num = 10;
tri_anx_time  = 5245.123456;
tri_orbit_duration = 6035.928144;

status = xl_time_ref_init(tri_time, &tri_orbit_num, &tri_anx_time,
                          &tri_orbit_duration, &time_id, xl_ierr);
if (status != XL_OK)
{
   func_id = XL_TIME_REF_INIT_ID;
   xl_get_msg(&func_id, xl_ierr, &n, msg);
   xl_print_msg(&n, msg);
   if (status <= XL_ERR) return(XL_ERR);
}
```

Time initialisation

```
/* Satellite Nominal Attitude frame initialisation */

model_enum = XP_MODEL_ENVISAT;
model_param[0] = -0.1671;
model_param[1] =  0.0501;
model_param[2] =  3.9130;

local_status = xp_sat_nominal_att_init_model(&model_enum, model_param,
                                             &sat_nom_trans_id, xp_ierr);
if (status != XP_OK)
{
   func_id =  XP_SAT_NOMINAL_ATT_INIT_MODEL_ID;
   xp_get_msg(&func_id, xp_ierr, &n, msg);
   xp_print_msg(&n, msg);
   if (status <= XP_ERR) return(XP_ERR);
}
```

Satellite Nominal Attitude frame

```
/* Satellite Attitude frame initialisation */
ang[0] = 0.0046941352;
ang[1] = 0.0007037683;
ang[2] = 356.09346792;

local_status = xp_sat_att_angle_init(ang, &sat_trans_id, xp_ierr);

if (status != XP_OK)
{
   func_id =  XP_SAT_ATT_ANGLE_INIT_ID;
   xp_get_msg(&func_id, xp_ierr, &n, msg);
   xp_print_msg(&n, msg);
   if (status <= XP_ERR) return(XP_ERR);
}
```

Satellite Attitude frame

```
/* attitude initialisation */

status = xp_attitude_init (&attitude_id, xp_ierr);

if (status != XL_OK)
{
   func_id = XP_ATTITUDE_INIT_ID;
   xp_get_msg(&func_id, xp_ierr, &n, msg);
   xp_print_msg(&n, msg);
}
```

Attitude Initialisation

```
/* Get attitude */
target_frame = XP_SAT_ATT;
time_ref = XL_TIME_UTC;     /* Satellite state vector */
time      = 255.3456;
pos[0]    =  6997887.57;
pos[1]    = -1536046.83;
pos[2]    =    99534.18;
vel[0]    =     -240.99;
vel[1]    =    -1616.85;
vel[2]    =    -7376.65;
acc[0]    =       -7.79104;
acc[1]    =        1.69353;
acc[2]    =       -0.10826;
local_status = xp_attitude_compute(&model_id, &time_id, &sat_nom_trans_id,
                                   &sat_trans_id, &instr_trans_id,
                                   &attitude_id, &time_ref, &time,
                                   pos, vel, acc, &target_frame, xp_ierr);
if (status != XP_OK)
{
   func_id =  XP_ATTITUDE_COMPUTE_ID;
   xp_get_msg(&func_id, xp_ierr, &n, msg);
   xp_print_msg(&n, msg);
   if (status <= XP_ERR) return(XP_ERR);
}
```

Attitude computation

```
/* Get attitude data */
status = xp_attitude_get_id_data(&attitude_id, &attitude_data);
printf("- Init Status      : %li\n", xp_attitude_init_status(&attitude_id));
printf("- Init Mode        : %li\n", xp_attitude_get_mode(&attitude_id));
printf("- Model            : %li\n", attitude_data.model);
printf("- Time Reference   : %li\n", attitude_data.time_ref);
printf("- Time             : %lf\n", attitude_data.time);
printf("- Sat Position     : [%12.3lf,%12.3lf,%12.3lf]\n",
       attitude_data.sat_vector.v[0],
       attitude_data.sat_vector.v[1],
       attitude_data.sat_vector.v[2]);
[...]
printf("- Source frame     : %lf"\n, attitude_data.source_frame);
printf("- Target frame     : %lf\n", attitude_data.target_frame);
printf("- Attitude Matrix  : %lf\t%lf%lf\n",
       attitude_data.sat_mat.m[0][0], attitude_data.sat_mat.m[0][1],
       attitude_data.sat_mat.m[0][2]);
printf("                     %lf\t%lf%lf\n",
       attitude_data.sat_mat.m[1][0], attitude_data.sat_mat.m[1][1],
       attitude_data.sat_mat.m[1][2]);
printf("                     %lf\t%lf%lf\n",
       attitude_data.sat_mat.m[2][0], attitude_data.sat_mat.m[2][1],
       attitude_data.sat_mat.m[2][2]);
[...]
```

Getting attitude data

```
    /* Get the attitude for a new satellite position
       Note that it is not necessary to close the attitude_id */
    target_frame = XP_SAT_ATT;
    time_ref = XL_TIME_UTC;     /* Satellite state vector */
    time     = 255.3456;
    pos[0]   =  4859964.138;
    pos[1]   = -5265612.059;
    pos[2]   =        0.002;
    vel[0]   =    -1203.303801;
    vel[1]   =    -1098.845511;
    vel[2]   =     7377.224410;
    acc[0]   =        0.0;
    acc[1]   =        0.0;
    acc[2]   =        0.0;
    local_status = xp_attitude_compute(&model_id, &time_id,
                                       &sat_nom_trans_id,
                                       &sat_trans_id, &instr_trans_id,
                                       &attitude_id, &time_ref, &time,
                                       pos, vel, acc, &target_frame,
                                        xp_ierr);
    if (status != XP_OK)
    {
       func_id =  XP_ATTITUDE_COMPUTE_ID;
       xp_get_msg(&func_id, xp_ierr, &n, msg);
       xp_print_msg(&n, msg);
       if (status <= XP_ERR) return(XP_ERR);
    }
```

Attitude computation

```
   /* Close attitude */
   status = xp_attitude_close(&attitude_id, xp_ierr);
   if (status != XL_OK)
   {
      func_id = XP_ATTITUDE_CLOSE_ID;
      xp_get_msg(&func_id, xp_ierr, &n, msg);
      xp_print_msg(&n, msg);
 }
```

Attitude Close

```
 /* Close Satellite Nominal Attitude frame */
 status = xp_sat_nominal_att_close(&sat_nom_trans_id, xp_ierr);
 if (status != XL_OK)
 {
    func_id = XP_SAT_NOMINAL_ATT_CLOSE_ID;
    xp_get_msg(&func_id, xp_ierr, &n, msg);
    xp_print_msg(&n, msg);
 }
```

Close Sat. Att. Nom.

```
/* Close Satellite Attitude frame */
status = xp_sat_att_close(&sat_trans_id, xp_ierr);
if (status != XL_OK)
{
   func_id = XP_SAT_ATT_CLOSE_ID;
   xp_get_msg(&func_id, xpierr, &n, msg);
   xp_print_msg(&n, msg);
}
```

Close Sat Att. frame

```
/* Close time_id */
status = xp_time_close(&time_id, xl_ierr);
if (status != XL_OK)
{
   func_id = XP_TIME_CLOSE_ID;
   xp_get_msg(&func_id, xl_ierr, &n, msg);
   xp_print_msg(&n, msg);
}
```

Close Time

Example 4.16 - II: Attitude defined by star tracker for cryosat.

```
/* Variables */
[...]
char att_file[] = "../../data/CRYOSAT_STAR_TRACKER_DATA.DBL";
char auxiliary_file[] = "../../data/cryosat_reference_frame_conf.xml";

[ ... Time initialisation... ]
```

Satellite attiude frame

```
/* satellite reference initialization */

files[0] = att_file;
n_files = 1;
time_init_mode = XO_SEL_FILE;
time_ref = XL_TIME_UTC;
time0 = 1646.50;
time1 = 1646.60;
target_frame = XP_SAT_ATT;

status = xp_sat_att_init_file(&time_id, &n_files, files, auxiliary_file,
                              &time_init_mode, &time_ref, &time0, &time1,
                              &val_time0, &val_time1,
                              &sat_trans_id, xp_ierr);
if (status != XL_OK)
{
   func_id = XP_SAT_ATT_INIT_FILE_ID;
   xp_get_msg(&func_id, xp_ierr, &n, msg);
   xp_print_msg(&n, msg);
}
```

Attitude Initialisation

```
/* attitude initialisation */
status = xp_attitude_init (&attitude_id, xp_ierr);
if (status != XL_OK)
{
   func_id = XP_ATTITUDE_INIT_ID;
   xp_get_msg(&func_id, xp_ierr, &n, msg);
   xp_print_msg(&n, msg);
}
```

```
/* attitude computation */
time = 1646.775;
pos[0]  = +2117636.668;
pos[1]  = -553780.175;
pos[2]  =  -6748229.578;
vel[0]  = +6594.65340;
vel[1]  = -2760.52030;
vel[2]  = +2303.10280;

status = xp_attitude_compute(&model_id, &time_id,
                             &sat_nom_trans_id,
                             &sat_trans_id, &instr_trans_id,
                             &attitude_id, &time_ref, &time,
                             pos, vel, acc, &target_frame,
                             xp_ierr);
if (status != XL_OK)
{
   func_id = XP_ATTITUDE_COMPUTE_ID;
   xp_get_msg(&func_id, xp_ierr, &n, msg);
   xp_print_msg(&n, msg);
}
```

Attitude Computation

```
  [... Attitude usage...]
```

```
/* Close attitude */
status = xp_attitude_close(&attitude_id, xp_ierr);
if (status != XL_OK)
{
   func_id = XP_ATTITUDE_CLOSE_ID;
   xp_get_msg(&func_id, xp_ierr, &n, msg);
   xp_print_msg(&n, msg);
}
```

Close Attitude

```
/* Close Satellite Attitude frame */
status = xp_sat_att_close(&sat_trans_id, xp_ierr);
if (status != XL_OK)
{
   func_id = XP_SAT_ATT_CLOSE_ID;
   xp_get_msg(&func_id, xp_ierr, &n, msg);
   xp_print_msg(&n, msg);
}
```

Close Attitude frame

```
  [ Close time_id ...]
```

4.16.3 Atmospheric initialisation

When using an atmospheric model, the ID *xp_atmos_id* structure should initialised by calling the CFI function **xp_atmos_init** (see [PNT_SUM]) providing the needed atmospheric model and files.

Once the *xp_atmos_id* has been initialised, it can be used as an input parameter for target calculations (see section 4.16.5).

The memory allocated for *xp_atmos_id* should be freed when the structure is not to be used in the program by calling the CFI function **xp_atmos_close**.

4.16.4 Digital Elevation model

Before using a digital elevation model, the ID *xp_dem_id* structure should initialised by calling the CFI function **xp_dem_init** (see [PNT_SUM]) providing the configuration file for the DEM.

Once the *xp_dem_id* has been initialised, it can be used as an input parameter for target calculations (see section 4.16.5).

The memory allocated for *xp_dem_id* should be freed when the structure is not to be used in the program by calling the CFI function **xp_dem_close**.

4.16.5 Targets

Once the attitude has been initialised and optionally have the atmospheric and the DEM models, the targets can be calculated. For this issue there is a set of functions that solves different types of pointing problems. A detailed explanation of the different target problems can be seen in [PNT_SUM] section 4.

For every target problem, three different target types are defined:

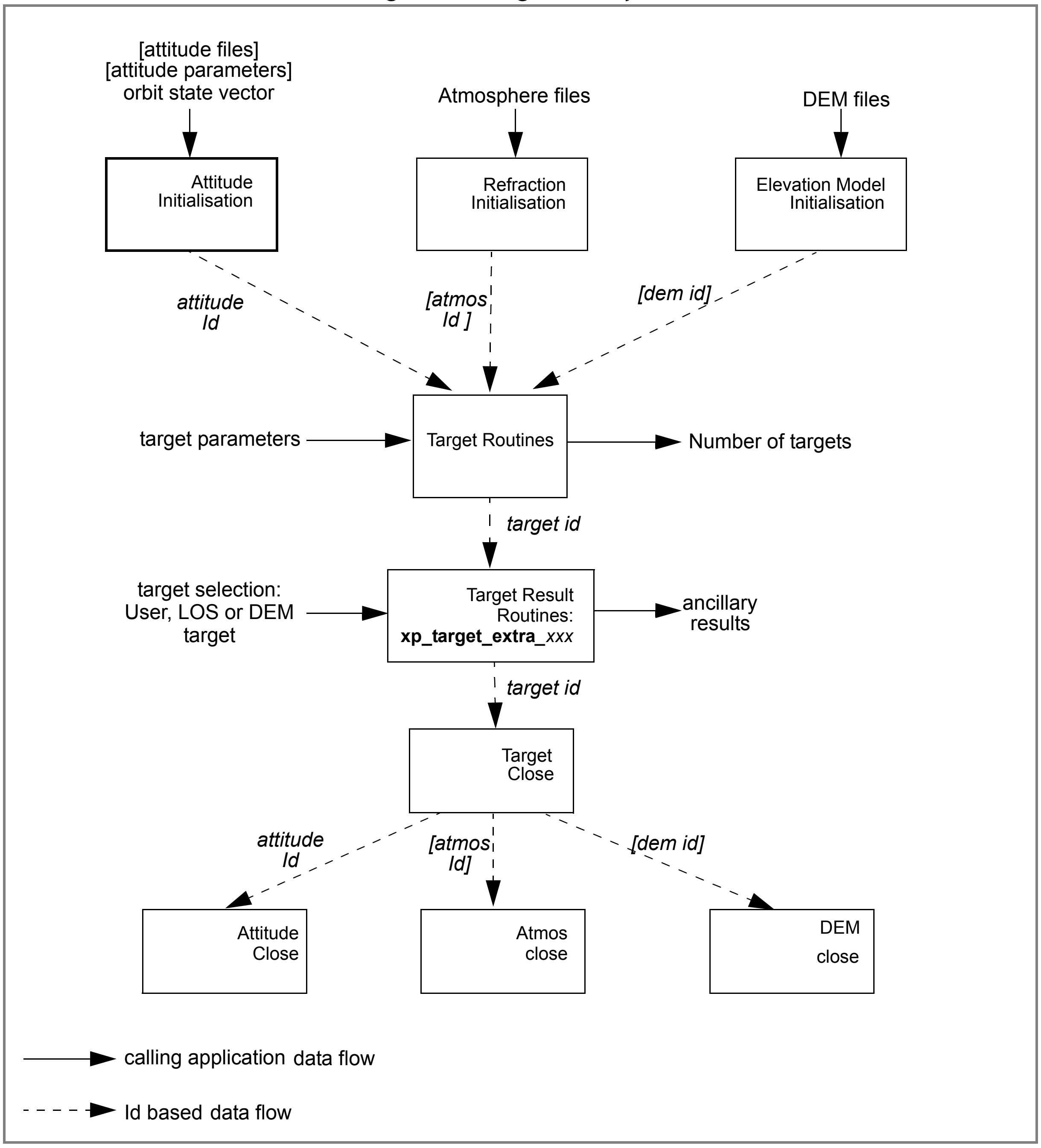
- User target: it is the target requested by the user.
- LOS target (line of sight target): it is the computed raypath to reach the user target.
- DEM target: it is a target computed taking into account the DEM model. It is only used for geolocated targets.

The previous functions do not return directly the computed target parameters, but another ID called *xp_target_id***.** The target data for one of the target types (user, LOS or DEM) has to be retrieved from the *xp_target_id* using another set of functions called **xp_target_extra**_*xxx*.

Once a target is not to be used any more, it has to closed in order to free internal memory by calling **xp_target_close**.

The following figure summarises the data flow for the target calculation:

Figure 9: Target data flow



Example 4.16 - III: Target Star.

```
/* Local Variables */
[...]

[ ... Time initialisation...]
```

Satellite Nominal Attiude frame

```
/* Satellite Nominal attitude frame initialisation */

sat_id         = XP_SAT_ENVISAT;
model_enum     = XP_MODEL_ENVISAT;
model_param[0] = -0.1671;
model_param[1] = 0.0501;
model_param[2] = 3.9130;

local_status = xp_sat_nominal_att_init_model(&model_enum, model_param,
                                             &sat_nom_trans_id, xp_ierr);
if (status != XP_OK)
{
   func_id = XP_SAT_NOMINAL_ATT_INIT_MODEL_ID;
   xp_get_msg(&func_id, xp_ierr, &n, msg);
   xp_print_msg(&n, msg);
   if (status <= XP_ERR) return(XP_ERR);
}
```

Attitude Initialisation

```
/* Attitude initialisation */

status = xp_attitude_init (&attitude_id, xp_ierr);
if (status != XP_OK)
{
   func_id = XP_ATTITUDE_INIT_ID;
   xp_get_msg(&func_id, xp_ierr, &n, msg);
   xp_print_msg(&n, msg);
   if (status <= XP_ERR) return(XP_ERR);
}
```

```
    /* Attitude computation */
    time_ref       = XL_TIME_UT1;
    time           = 255.3456;
    pos[0]         =  4859964.138;
    pos[1]         = -5265612.059;
    pos[2]         =        0.002;
    vel[0]         =    -1203.303801;
    vel[1]         =    -1098.845511;
    vel[2]         =     7377.224410;
    acc[0]         =        0.0;
    acc[1]         =        0.0;
    acc[2]         =        0.0;

    target_frame  = XP_SAT_NOM_ATT;

    status = xp_attitude_compute(&model_id, &time_id, &sat_nom_trans_id,
                                 &sat_trans_id, &instr_trans_id, &attitude_id,
                                 &time_ref, &time, pos, vel, acc,
                                 &target_frame, xp_ierr);
    if (status != XP_OK)
    {
       func_id = XP_ATTITUDE_COMPUTE_ID;
       xp_get_msg(&func_id, xp_ierr, &n, msg);
       xp_print_msg(&n, msg);
       if (status <= XP_ERR) return(XP_ERR);
    }
```

```
/* Call xp_target_star function */

deriv         =    XL_DER_1ST;
star_ra       =     272.0;
star_dec      =     -73.0;
star_ra_rate  =       0.0;
star_dec_rate =       0.0;
freq          =       1.e10;

status = xp_target_star(&sat_id, &attitude_id, &atmos_id, &dem_id,
                        &deriv, &star_ra, &star_dec,
                        &star_ra_rate, &star_dec_rate, &iray, &freq,
                        &num_user_target, &num_los_target,
                        &target_id, xp_ierr);
if (status != XP_OK)
{
   func_id = XP_TARGET_STAR_ID;
   xp_get_msg(&func_id, xp_ierr, &n, msg);
   xp_print_msg(&n, msg);
   if (status <= XP_ERR) return(XP_ERR);
}
```

```
/* Get user target parameters from the target_id */

choice        = XL_DER_1ST;
target_type   = XP_USER_TARGET_TYPE;
target_number = 0;

status = xp_target_extra_vector(&target_id, &choice,
                                &target_type, &target_number,
                                results, results_rate,
                                results_rate_rate, xp_ierr);
if (status != XP_OK)
{
   func_id = XP_TARGET_EXTRA_VECTOR_ID;
   xp_get_msg(&func_id, xp_ierr, &n, msg);
   xp_print_msg(&n, msg);
   if (status <= XP_ERR) return(XP_ERR);
}

/* Print results */
printf("  OUTPUT \n");
printf("- Target Position  : [%12.3lf,%12.3lf,%12.3lf]",
       results[0], results[1], results[2]);
printf("- Target Velocity  : [%12.3lf,%12.3lf,%12.3lf]",
       results_rate[0], results_rate[1], results_rate[2]);
printf("- Range            : %lf", results[6]);
printf("- Range Rate       : %lf", results_rate[6]);
printf("- Sat-Target LOS   : [%12.9lf,%12.9lf,%12.9lf]",
          results[3], results[4], results[5]);
printf("- Sat-Tar LOS Rate : [%12.9lf,%12.9lf,%12.9lf]",
          results_rate[3], results_rate[4], results_rate[5]);
[...]
```

Using target

```
/* Closing Ids */
status = xp_target_close(&target_id, xp_ierr);
{
   func_id = XP_TARGET_CLOSE_ID;
   xp_get_msg(&func_id, xp_ierr, &n, msg);
   xp_print_msg(&n, msg);
```

Close target

```
    status = xp_attitude_close(&attitude_id, xp_ierr);
    {
       func_id = XP_ATTITUDE_CLOSE_ID;
       xp_get_msg(&func_id, xp_ierr, &n, msg);
       xp_print_msg(&n, msg);
    }
```

Close attitude

```
    status = xp_sat_nominal_att_close(&sat_nom_trans_id, xp_ierr);
    {
       func_id = XP_SAT_NOMINAL_ATT_CLOSE;
       xp_get_msg(&func_id, xp_ierr, &n, msg);
       xp_print_msg(&n, msg);
    }
```

Close Sat. Nom.

```
    [ Close time initialisation...]
```

Example 4.16 - IV: Target intersection: target computation along one orbit.

The following code shows a complete example for:

- time initialisation
- Orbit initialisation
- Attitude initialisation
- Getting the intersection target for different points along one orbit

[...]

```
/* Local variables declaration */
long   status;
long   n;
long   func_id;
char   msg[XL_MAX_COD][XL_MAX_STR];
long   xl_ierr[XL_ERR_VECTOR_MAX_LENGTH];
long   xo_ierr[XO_ERR_VECTOR_MAX_LENGTH];
long   xp_ierr[XP_ERR_VECTOR_MAX_LENGTH];

long             sat_id;
xl_model_id      model_id     = {NULL};
xl_time_id       time_id      = {NULL};
xo_orbit_id      orbit_id     = {NULL};
xo_propag_id     propag_id    = {NULL};

xp_sat_nom_trans_id sat_nom_trans_id = {NULL};
xp_sat_trans_id     sat_trans_id     = {NULL};
xp_instr_trans_id   instr_trans_id   = {NULL};
xp_attitude_id      attitude_id      = {NULL};

xp_atmos_id         atmos_id         = {NULL};
xp_dem_id           dem_id           = {NULL};
xp_target_id        target_id        = {NULL};

[...]
```

Varaible declaration

Time initialisation

```
/* Time initialization */

 time_model      = XL_TIMEMOD_FOS_PREDICTED;
 n_files         = 1;
 time_init_mode = XL_SEL_FILE;
 time_ref        = XL_TIME_UTC;
 time0           = 0;
 time1           = 0;
 orbit0          = 0;
 orbit1          = 0;
 time_file[0] = orbit_file;

 status = xl_time_ref_init_file(&time_model, &n_files, time_file,
                                &time_init_mode, &time_ref, &time0, &time1,
                                &orbit0, &orbit1, &val_time0, &val_time1,
                                &time_id, xl_ierr);
 if (status != XL_OK)
 {
   func_id = XL_TIME_REF_INIT_FILE_ID;
   xl_get_msg(&func_id, xo_ierr, &n, msg);
   xl_print_msg(&n, msg);
   if (status <= XL_ERR) return(XL_ERR);
 }
```

Orbit initialisation

```
 /* Orbit initialization */
 time_init_mode = XO_SEL_FILE;

 input_files[0] = orbit_file;
 n_files = 1;
 orbit_mode = XO_ORBIT_INIT_AUTO;

 status =  xo_orbit_init_file(&sat_id, &model_id, &time_id,
                              &orbit_mode, &n_files, input_files,
                              &time_init_mode, &time_ref_utc,
                              &time0, &time1, &orbit0, &orbit1,
                              &val_time0, &val_time1, &orbit_id,
                              xo_ierr);
 if (status != XO_OK)
 {
   func_id = XO_ORBIT_INIT_FILE_ID;
   xo_get_msg(&func_id, xo_ierr, &n, msg);
   xo_print_msg(&n, msg);
   xl_time_close(&time_id, xl_ierr);
   if (status <= XL_ERR) return(XL_ERR);
 }
```

```
/* Satellite Nominal Attitude frame initialisation */

/* Yaw Steering Mode */
model_enum = XP_MODEL_GENERIC;
model_param[0] = XP_NEG_Z_AXIS;
model_param[1] = XP_NADIR_VEC;
model_param[2] = 0.;
model_param[3] = 0.;
model_param[4] = 0.;
model_param[5] = XP_X_AXIS;
model_param[6] = XP_EF_VEL_VEC;
model_param[7] = 0.;
model_param[8] = 0.;
model_param[9] = 0.;
status = xp_sat_nominal_att_init_model(&model_enum, model_param,
                                      /* output */
                                      &sat_nom_trans_id, xp_ierr);
if (status != XP_OK)
{
  func_id = XP_SAT_NOMINAL_ATT_INIT_MODEL_ID;
  xp_get_msg(&func_id, xp_ierr, &n, msg);
  xp_print_msg(&n, msg);
  xo_propag_close(&propag_id, xo_ierr);
  xo_orbit_close(&orbit_id, xo_ierr);
  xl_time_close(&time_id, xl_ierr);
  if (status <= XO_ERR) return(XL_ERR);
}
```

```
/* Satellite Attitude frame initialisation */

  ang[0] =  0.0;
  ang[1] =  0.0;
  ang[2] =  0.0;
  status = xp_sat_att_angle_init(ang,
                               /* output */
                               &sat_trans_id,
                               xp_ierr);
  if (status != XP_OK)
  {
    func_id = XP_SAT_ATT_ANGLE_INIT_ID;
    xp_get_msg(&func_id, xp_ierr, &n, msg);
    xp_print_msg(&n, msg);
    xp_sat_nominal_att_close(&sat_nom_trans_id, xp_ierr);
    xo_propag_close(&propag_id, xo_ierr);
    xo_orbit_close(&orbit_id, xo_ierr);
    xl_time_close(&time_id, xl_ierr);
    if (status <= XO_ERR) return(XL_ERR);
  }
```

Satellite Attitude initialisation

```
 /* Instrument attitude frame initailisation */

  ang[0] =  0.0;
  ang[1] =  0.0; /* scan angle */
  ang[2] =  0.0;

  offset[0] = 0.0;
  offset[1] = 0.0;
  offset[2] = 0.0;

  status = xp_instr_att_angle_init(ang, offset,
                                 /* output */
                                 &instr_trans_id,
                                 xp_ierr);
  if (status != XP_OK)
  {
    [...]
  }
```

Instrument Attitude initialisation

```
/* Attitude initialisation */
 status = xp_attitude_init (&attitude_id, xp_ierr);
 if (status != XP_OK)
 {
   [...]
 }
```

Attitude initialisation

```
/* DEM initialisation */
 dem_mode = XD_DEM_ACE_MODEL;
 status = xp_dem_init(&dem_mode, &dem_model, dem_file,
                      &dem_id, xp_ierr);
 if (status != XP_OK)
 {
   [...]
 }
```

DEM initialisation

```
/* propagate along one orbit */

user_time_start = 2831.00690124781;
user_time_stop =  2831.07112143130;

time_step = 500/86400.0;

for (i_loop = user_time_start; i_loop < user_time_stop; i_loop += time_step)
{
  time = i_loop;

  /* Get satellite state vector at "time" */
  status = xo_osv_compute(&orbit_id, &propag_model, &time_ref_utc, &time,
                          pos, vel, acc, xo_ierr);
  if (status != XP_OK)
  {
    [...]
  }
```

Loop to get targets for different times between user_time_start and user_time_stop

Compute State vector

```
/* Compute Attitude using the calculated state vector */

  target_frame  = XP_INSTR_ATT;

  status = xp_attitude_compute(&model_id, &time_id,
                               &sat_nom_trans_id,
                               &sat_trans_id,
                               &instr_trans_id,
                               &attitude_id,
                               &time_ref, &time,
                               pos, vel, acc,
                               &target_frame,
                               xp_ierr);
  if (status != XP_OK)
  {
    [...]
  }
```

Compute the attitude

```
/* Get the intersection target */

sat_id        = XP_SAT_ADM;
inter_flag    = XP_INTER_1ST;
deriv         = XL_DER_1ST;
los_az        =       90.0;
los_el        =       90.0;
los_az_rate   =        1.0;
los_el_rate   =        1.0;
iray          = XP_NO_REF;
freq          = 8.4e14; /* 355 nm, SPEED_OF_LIGHT = 299792458.0; [m/s] */
geod_alt = 0.0;
num_target    = 0;

status =  xp_target_inter(&sat_id,
                   &attitude_id,
                   &atmos_id,
                   &dem_id,
                   &deriv, &inter_flag, &los_az, &los_el,
                   &geod_alt,
                   &los_az_rate, &los_el_rate,
                   &iray, &freq,
                   /* output */
                   &num_user_target, &num_los_target,
                   &target_id,
                   xp_ierr);
if (status != XP_OK)
{
  [...]
}
```

Compute the target

```
/* Get User, LOS and DEM  Targets Data */

   for (target_type    = XP_USER_TARGET_TYPE;
     target_type <= XP_DEM_TARGET_TYPE;
     target_type++)
   {
     if (target_type == XP_USER_TARGET_TYPE)
        strcpy(target_name, "User target");
     if (target_type == XP_LOS_TARGET_TYPE)
        strcpy(target_name, "LOS target");
     else if (target_type == XP_DEM_TARGET_TYPE)
        strcpy(target_name, "DEM target");

    printf("\n-----------------------------------------------------------\n");
    printf("  Target results for xp_target_inter and target %d\n", target_number);
    printf("  Target type: %s. Time = %f\n", target_name, time);
    printf("-------------------------------------------------------------\n");

    target_number = 1;
    choice        = XL_DER_1ST;

    /* Get target parameters */
    status = xp_target_extra_vector(&target_id,
                                    &choice, &target_type, &target_number,
                                    /* output */
                                    vector_results,
                                    vector_results_rate,
                                    vector_results_rate_rate,
                                    xp_ierr);
     if (status == XP_ERR)
     {
        [...]
     }
     else
     {
    printf("\n  Target extra results \n");
    printf("- Num Target        : %ld\n", targ_num);
    printf("- Target Position   : [%12.3lf,%12.3lf,%12.3lf]\n",
         vector_results[0], vector_results[1], vector_results[2]);
    printf("- Target Velocity   : [%12.3lf,%12.3lf,%12.3lf]\n",
          vector_results_rate[0], vector_results_rate[1], vector_results_rate[2]);
    printf("- Range             : %lf\n",vector_results[6]);
      [...]
    }
```

Loop to get data for the different targets

Getting target data

```
    /* Get target extra main parameters */

    choice = XP_TARG_EXTRA_AUX_ALL;
    status = xp_target_extra_main(&target_id,
                                  &choice, &target_type, &target_number,
                                  main_results, main_results_rate,
                                   main_results_rate_rate,
                                  xp_ierr);
    if (status == XP_ERR)
    {
     [...]
    }
    else
    {
   printf("\n  Target extra results \n");
   printf("- Num Target    : %ld\n", targ_num);
   printf("- Geocentric Long.          : %lf\n",main_results[0]);
   printf("- Geocentric Lat.           : %lf\n",main_results[1]);
   printf("- Geodetic Latitude         : %lf\n",main_results[2]);
      [...]
   }
    /* Get target extra results */
    choice        = XP_TARG_EXTRA_AUX_ALL;
    target_number = 0;
    status = xp_target_extra_aux(&target_id,
                              &choice, &target_type, &target_number,
                              aux_results, aux_results_rate, aux_results_rate_rate,
                               xp_ierr);
    if (status == XP_ERR)
   {
      [...]
   }
    else
    {
      printf("\n  Auxiliary Target outputs:\n");
      printf("- Curvature Radius at target's nadir = %lf\n", aux_results[0]);
      printf("- Distance: target's nadir to satellites's nadir = %lf\n",
                   aux_results[1]);
      printf("- Distance target's nadir to ground track = %lf\n",
              aux_results[2]);
      printf("- Distance SSP to point in the ground track nearest to the target's
nadir= %lf\n", aux_results[3]);
      printf("- MLST at target = %lf\n", aux_results[4]);
      printf("- TLST at target = %lf\n", aux_results[5]);
      printf("- RA throught the atmosphere = %lf\n", aux_results[6]);
      [...]
    }
    /* Get target-to-sun parameters */
    choice        = XL_DER_1ST;
    target_number = 0;
    iray          = XP_NO_REF;
    freq          =        1.e10;

    status = xp_target_extra_target_to_sun(&target_id,
                                   &choice, &target_type, &target_number,
                                        &iray, &freq,
                                        sun_results, sun_results_rate,
                                        sun_results_rate_rate, xp_ierr);
    if (status == XP_ERR)
    {
      [...]
    }
    else
{
      printf("\n  Target to Sun outputs:\n");
      printf("- Topocentric Azimuth.        : %lf\n",sun_results[0]);
      printf("- Topocentric Elevation.      : %lf\n",sun_results[1]);
      printf("- Topocentric Azimuth rate.   : %lf\n",sun_results_rate[0]);
      printf("- Topocentric Elevation rate  : %lf\n",sun_results_rate[1]);
      printf("- Tangent Altitude            : %lf\n",sun_results[2]);
      printf("- Target to sun visibility.   : %g\n",sun_results[3]);
     }
    /* Get target-to-moon parameters */
    choice        = XL_DER_1ST;
    target_number = 0;
    iray          = XP_NO_REF;
```

```
      /* Get EF target parameters */
      choice        = XL_DER_1ST;
      target_number = 0;
      freq          =       1.e10;
      status = xp_target_extra_ef_target(&target_id,
                              &choice, &target_type, &target_number, &freq,
                              ef_target_results_rate,
                             ef_target_results_rate_rate,
                             xp_ierr);
      if (status == XP_ERR)
      {
        [...]
      }
      else
      {
     printf("\n EF Target outputs:\n");
     printf("- EF target to satellite range rate : %lf\n",
                 ef_target_results_rate[1]);
     printf("- EF target to satellite azimuth rate (TOP) : %lf\n",
                 ef_target_results_rate[2]);
     printf("- EF target to satellite elevation rate (TOP) : %lf\n",
                 ef_target_results_rate[3]);
        [...]
      }

    } /* end for "target_type" (End loop to get data for the different targets)*/
```

Getting the targe

```
    /* Closing Ids */
    status = xp_target_close(&target_id, xp_ierr);
    [...]
} /* end for "i_loop" (End loop for different times )*/
```

Closing target

```
  status = xp_attitude_close(&attitude_id, xp_ierr);
  [...]
```

Closing attitude

```
  status = xp_sat_nominal_att_close(&sat_nom_trans_id, xp_ierr);
  [...]

  status = xp_sat_att_close(&sat_trans_id, xp_ierr);
  [...]

  status = xp_instr_att_close(&instr_trans_id, xp_ierr);
  [...]
```

Closing satellite attitude frames

```
  status = xp_dem_close(&dem_id, xp_ierr);
  [...]
```

Closing DEM

```
  status = xo_orbit_close(&orbit_id, xo_ierr);
  [...]
```

Closing orbit

```
  status = xl_time_close(&time_id, xl_ierr);
  [...]
```

Closing time

```
}

/* end */
```

4.17 Swath calculations

A swath can be defined as the track swept by the field of view of an instrument in the satellite along a time interval. For the aim of this section this definition is enough, however the definition of a swath can be much more complex. For a detailed definition about swaths refer to [VIS_SUM] section 7.1.2.

The Earth Explorer CFI software can handle swath data using two different data sets provided by :

- Swath Definition files (SDF): they contain information about the swath type and geometry and the satellite attitude.

or

- Swath Template files (STF): they contain the list of longitude and latitude points of the swath if the orbit started at longitude and latitude 0.

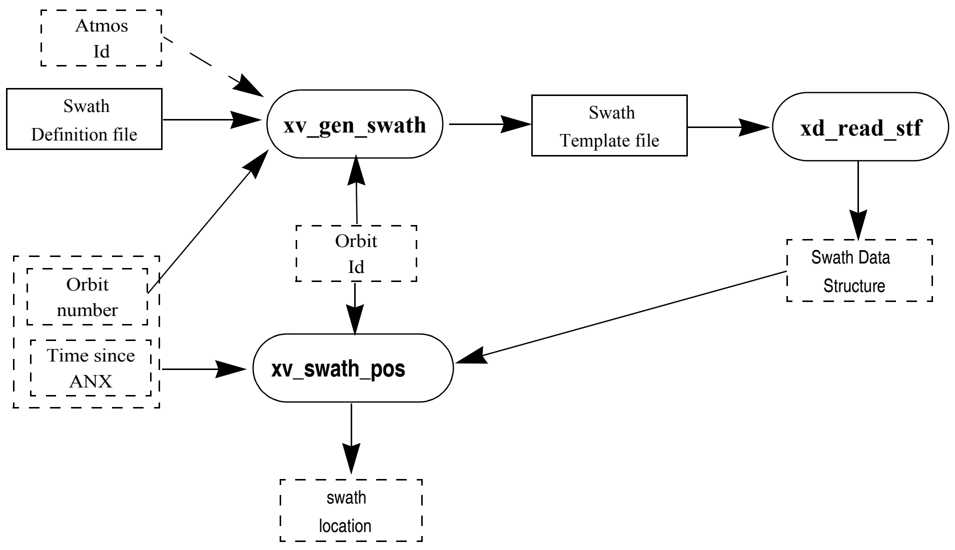
The format for the two files can be found in [D_H_SUM].

Swath files are mainly useful for the the visibility calculations (section 4.18) but the CFI software provides other functions for getting information from swaths:

- Reading and writing swath files (see section 4.3, 4.4 and [D_H_SUM]): These functions allow the user to read a swath file and store the information in a data structure (reading functions) or to dump to a file the swath data contained in a structure (writing function).
- Generate a STF from a SDF (function **xv_gen_swath** described in [VIS_SUM]): this operation requieres the initialisation of the *xo_orbit_id* (section 4.12) and optionally the *atmos_id* (4.16.3) if the swath has to take into account the raypath refraction by the atmosphere.
- Calculating the swath position for a given time (function **xv_swath_pos** described in [VIS_SUM]): This operation requieres the initialisation of the *xo_orbit_id* and the data structure containing the swath points from a STF (read with **xd_read_stf**).

The following figure shows an schema for the calling sequence for the described operations:

Figure 10: EXPLORER_VISIBILITY Data Flow



Note that in order to produce consistent data the same *xo_orbit_id* is used in the two calls of the swath functions.

Also the orbit number introduced in **xv_gen_swath** is the same orbit number that is passed to **xv_swath_pos**. This is not mandatory but advisable. **xv_gen_swath** produce the STF taken into account the orbit geometry so it produces the same file for all orbits with the same geometry (for example, all the orbits within the same orbital change in an OSF). In consequence, there is not need of generating a new STF every time that **xv_swath_pos** is going to be called for a different orbit, only it is needed if the orbit geometry changes.

Example 4.17 - I: Getting the swath position.

```
/* Variables */
long   status, func_id, n;
char   msg[XL_MAX_COD][XL_MAX_STR];
long   xl_ierr[XL_ERR_VECTOR_MAX_LENGTH];
long   xd_ierr[XD_ERR_VECTOR_MAX_LENGTH];
long   xo_ierr[XO_ERR_VECTOR_MAX_LENGTH];
long   xv_ierr[XV_ERR_VECTOR_MAX_LENGTH];

long         sat_id     = XO_SAT_CRYOSAT;
xl_time_id  time_id    = {NULL};
xl_model_id model_id   = {NULL};
xo_orbit_id orbit_id   = {NULL};
xp_atmos_id atmos_id   = {NULL};

double tri_time[4],
       tri_orbit_num = 10, /* dummy */
       tri_anx_time   = 5245.123456,/* dummy */
       tri_orbit_duration = 6035.928144; /* dummy */

long n_files, time_mode, orbit_mode, time_ref;
char orbit_scenario_file[XD_MAX_STR];
char *files[2];

long req_orbit;
char dir_name[256];
char sdf_name[256], stf_name[256];
char file_class[] = "TEST";
long version_number = 1;
char fh_system[] = "CFI";

xd_stf_file stf_data;
long orbit_type, abs_orbit, second, microsec, cycle;
double long_swath, lat_swath, alt_swath;
```

Variable declaration

```
     /* Time initialisation */
     tri_time[0] = -245.100000000;            /* TAI time [days] */
     tri_time[1] = tri_time[0] - 35.0/86400.; /* UTC time [days] (= TAI - 35.0 s) */
     tri_time[2] = tri_time[0] - 35.3/86400.; /* UT1 time [days] (= TAI - 35.3 s) */
     tri_time[3] = tri_time[0] - 19.0/86400.; /* GPS time [days] (= TAI - 19.0 s) */

     status = xl_time_ref_init(tri_time, &tri_orbit_num, &tri_anx_time,
                               &tri_orbit_duration, &time_id, xl_ierr);
     [ ...error handling for xl_time_ref_init...]
```

Time initialisation

```
     /* Orbit initialisation: xo_orbit_init_file */
     n_files = 1;
     time_mode = XO_SEL_FILE;
     orbit_mode = XO_ORBIT_INIT_OSF_MODE;
     time_ref = XO_TIME_UT1;
     strcpy(orbit_scenario_file, "./CRYOSAT_XML_OSF");
     files[0] = orbit_scenario_file;

     status = xo_orbit_init_file(&sat_id, &model_id, &time_id,
                                 &orbit_mode, &n_files, files,
                                 &time_mode, &time_ref,
                                 &time0, &time1, &orbit0, &orbit1,
                                 &val_time0, &val_time1,
                                 &orbit_id, xo_ierr);
     [ ...error handling for xo_orbit_init_file...]
```

Orbit initialisation

```
     /* Generate Swath Template file */

     req_orbit = 150;
     strcpy(sdf_name, "./SDF_MERIS.EEF"); /* SDF */
     strcpy(dir_name, ""); /* -> generate file in current directory */
     strcpy(stf_name, "EXAMPLE_STF.EEF");

     status = xv_gen_swath(&orbit_id, &atmos_id, &req_orbit,
                           sdf_name, dir_name, stf_name,
                           file_class, &version_number, fh_system,
                           xv_ierr);
     if (status != XV_OK)
     {
        func_id = XV_GEN_SWATH_ID;
        xv_get_msg(&func_id, xv_ierr, &n, msg);
        xv_print_msg(&n, msg);
     }
```

Generate Swath Tmeplate file

```
/* Read Swath Template file */
status = xd_read_stf(stf_name, &stf_data, xd_ierr);
if (status != XV_OK)
{
   func_id = XD_READ_STF_ID;
   xd_get_msg(&func_id, xd_ierr, &n, msg);
   xv_print_msg(&n, msg);
}
```

Read STF

```
orbit_type = XV_ORBIT_ABS;
abs_orbit = 2950;
second = 100;
microsec = 500000;

status=xv_swath_pos(&orbit_id, &stf_data,
                    &orbit_type, &abs_orbit, &second, &microsec, &cycle,
                    &long_swath, &lat_swath, &alt_swath,
                    xv_ierr);
if (status != XV_OK)
{
   func_id = XV_SWATH_POS_ID;
   xv_get_msg(&func_id, xv_ierr, &n, msg);
   xv_print_msg(&n, msg);
}

/* print outputs */
printf("Input absolute Orbit + time since ANX: %d + %lf s\n",
        abs_orbit, second+(microsec*1.e-6));
printf("Outputs: \n");
printf("Swath point (longitude, latitude, altitude): (%lf, %lf, %lf) \n",
        long_swath, lat_swath, alt_swath);
```

Get swath position

```
/* free memory for the STF */
xv_free_stf(&stf_data);
```

Free STF

```
/* Close orbit_id */
status = xo_orbit_close(&orbit_id, xo_ierr);
if (status != XO_OK)
{
   func_id = XO_ORBIT_CLOSE_ID;
   xo_get_msg(&func_id, xo_ierr, &n, msg);
   xo_print_msg(&n, msg);
}
```

Closing orbit

```
        /* close time reference */
        status = xl_time_close(&time_id, xl_ierr);
        if (status != XO_OK)
        {
           func_id = XL_TIME_CLOSE_ID;
           xo_get_msg(&func_id, xl_ierr, &n, msg);
           xo_print_msg(&n, msg);
        }
```

Closing Time

```
       [...]
```

4.18 Visibility calculations

The Earth Explorer CFI software contains a set of functions to compute the time intervals in which a satellite instrument has visiblity of :

- an Earth zone
- a ground station
- a data relay satellite (DRS)
- a star

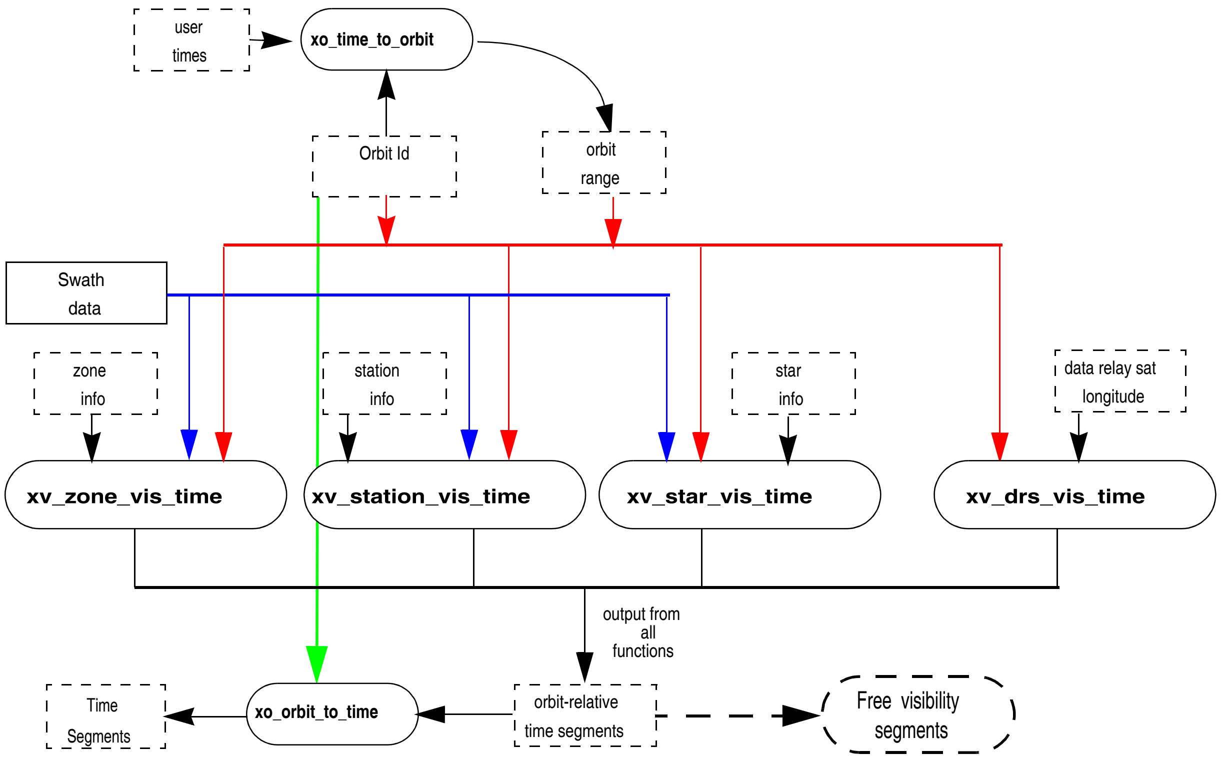
Visibility segments are provided as an orbit number plus the time since the ANX.

In order to calculate the visibility time intervals the functions requiere as inputs:

- orbital information provided via an orbit Id (see section 4.12)
- requested orbit interval in which the visibilities are to be computed.
- Swath information except for the DRS visibiility: It can be a swath definition file or a swath template file.
- Information about the target: zone, station, DRS or the star.

The following figure shows a possible calling sequence for visibility calculation:

Figure 11: EXPLORER_VISIBILITY Data Flow



Details about the visibility functions can be found in [VIS_SUM].

For those functions that require swath data, note that it can be provided by a SDF or a STF. The file type has to be indicated with an input flag (swath_flag):

- if swath_flag is zero, then the input file is a STF. Visibility segments will be computed with that file for all the requested orbits.
- if swath flag is greater than zero, then the input file is a SDF. The function will compute automatically the swath points. Thera are two possibilities:
  - The input *xo_orbit_id* was generated with an orbit scenario file or with **xo_orbit_init_def**: the swath points are generated only once for the first requested orbit. The visibility segments are computed with those swath points for all the orbits.
  - The input *xo_orbit_id* was generated with orbit state vectors: the swath points are generated for every n orbits, where n is the value of the swath_flag variable.

All the visibility functions return the segments as dynamical arrays, so when they are not to be used any more, the arrays should be freed.

Example 4.18 - I: Getting visibility segments for a zone.

```
/* Variables */
long   status, func_id, n;
char   msg[XL_MAX_COD][XL_MAX_STR];
long   xl_ierr[XL_ERR_VECTOR_MAX_LENGTH];
long   xo_ierr[XO_ERR_VECTOR_MAX_LENGTH];
long   xv_ierr[XV_ERR_VECTOR_MAX_LENGTH];

long        sat_id     = XO_SAT_CRYOSAT;
xl_time_id  time_id    = {NULL};
xl_model_id model_id   = {NULL};
xo_orbit_id orbit_id   = {NULL};

[... variables for time and orbit initialisation...]

long orbit_type, start_orbit, stop_orbit,
     start_cycle, stop_cycle;
long swath_flag;
char swath_file[256];
char zone_id[9], zone_db_file[XV_MAX_STR];
long projection, zone_num;
double zone_long[10], zone_lat[10], zone_diam;
long number_segments;
long *bgn_orbit, *bgn_second, *bgn_microsec, *bgn_cycle,
     *end_orbit, *end_second, *end_microsec, *end_cycle,
     *coverage;
```

Variable declaration

```
/* Time initialisation */
tri_time[0] = -245.100000000;            /* TAI time [days] */
tri_time[1] = tri_time[0] - 35.0/86400.; /* UTC time [days] (= TAI - 35.0 s) */
tri_time[2] = tri_time[0] - 35.3/86400.; /* UT1 time [days] (= TAI - 35.3 s) */
tri_time[3] = tri_time[0] - 19.0/86400.; /* GPS time [days] (= TAI - 19.0 s) */

status = xl_time_ref_init(tri_time, &tri_orbit_num, &tri_anx_time,
                          &tri_orbit_duration, &time_id, xl_ierr);
[ ...error handling for xl_time_ref_init...]
```

Time initialisation

```
/* Orbit initialisation: xo_orbit_init_file */
n_files = 1;
time_mode = XO_SEL_FILE;
orbit_mode = XO_ORBIT_INIT_OSF_MODE;
time_ref = XO_TIME_UT1;
strcpy(orbit_scenario_file, "./CRYOSAT_XML_OSF");
files[0] = orbit_scenario_file;

status = xo_orbit_init_file(&sat_id, &model_id, &time_id,
                            &orbit_mode, &n_files, files,
                            &time_mode, &time_ref,
                            &time0, &time1, &orbit0, &orbit1,
                            &val_time0, &val_time1,
                            &orbit_id, xo_ierr);
[ ...error handling for xo_orbit_init_file...]
```

Orbit initialisation

```
/* Calling xv_zone_vis_time */
orbit_type = XV_ORBIT_ABS;
start_orbit = 2900;
stop_orbit = 2950;

strcpy(swath_file, "./RA_2_SDF_.N1"); /* SDF */
strcpy(zone_id, "ZANA____");
strcpy (zone_db_file, "./ZONE_FILE.EEF");

projection = 0;
zone_num  = 0;    /* To be able to introduce the zone identifications */
min_duration = 0.0;

status = xv_zone_vis_time(&orbit_id, &orbit_type,
                          &start_orbit, &start_cycle,
                          &stop_orbit, &stop_cycle,
                          &swath_flag, swath_file,
                          zone_id, zone_db_file,
                          &projection, &zone_num,
                          zone_long, zone_lat, &zone_diam,
                          &min_duration,
                          &number_segments,
                          &bgn_orbit, &bgn_second,
                          &bgn_microsec, &bgn_cycle,
                          &end_orbit, &end_second,
                          &end_microsec, &end_cycle,
                          &coverage, xv_ierr);
if (status != XV_OK)
{
   func_id = XV_ZONE_VIS_TIME_ID;
   xv_get_msg(&func_id, xv_ierr, &n, msg);
   xv_print_msg(&n, msg);
}
```

```
    /* print outputs */
    printf("Inputs: \n");
    printf("  Start/Stop Absolute Orbit: %d / %d\n", start_orbit, stop_orbit);
    printf("   Zone File: %s\n", zone_db_file);
    printf("   Zone Id: %s\n", zone_id);
    printf("Outputs: \n");
    printf("Number of segments: %d\n", number_segments);
    printf("   Segments: Start (Orbit, seconds, microseconds) --
            Stop (Orbit, seconds, microseconds)\n");

    for(i=0; i < number_segments; i++)
    {
       printf("                (%4d, %4d, %6d) -- (%4d, %4d, %6d)\n",
               bgn_orbit[i], bgn_second[i], bgn_microsec[i],
               end_orbit[i], end_second[i], end_microsec[i]);
    }
```

Print visibility segments

```
    /* free memory: The cycle are not allocated as the orbit type
    is absolute orbits*/
    free(bgn_orbit);
    free(bgn_secs);
    free(bgn_microsecs);

    free(end_orbit);
    free(end_second);
    free(end_microsec);
    free (coverage);
```

Free memory for the visibility segments

```
    /* Close orbit_id */
    status = xo_orbit_close(&orbit_id, xo_ierr);
    if (status != XO_OK)
    {
       func_id = XO_ORBIT_CLOSE_ID;
       xo_get_msg(&func_id, xo_ierr, &n, msg);
       xo_print_msg(&n, msg);
    }
```

Closing orbit

```
/* close time reference */
status = xl_time_close(&time_id, xl_ierr);
if (status != XO_OK)
{
   func_id = XL_TIME_CLOSE_ID;
   xo_get_msg(&func_id, xl_ierr, &n, msg);
   xo_print_msg(&n, msg);
}
```

Closing time

```
  [...]
```

4.19 Time segments manipulation

The EXPLORER_VISIBILITY library provides a set of functions for doing logical operations between sets of time segments. A time segment is given by an absolute or relative orbit number plus the time since the ANX for the entry and the exit of the segment, this way the functions can handle the segments coming from the output of the visibility functions.

These operations are:

- Getting the complemet of a list of time segments (**xv_time_segments_not)**.
- Getting the intersection of two lists of time segments (**xv_time_segments_and**).
- Getting the union of two lists of time segments (**xv_time_segments_or**)
- Adding or subtracting time durations at the beginning and end of every time segment within a list (**xv_time_segments_delta**).
- Sorting a list of time segments (**xv_time_segments_sort**).
- Merging all the overlapped segments in a list (**xv_time_segments_merge**).
- Getting a subset of the time segments list, such that this subset covers entirely a zone or line swath (**xv_time_segments_mapping**).

A detailed explanation of these functions is in [VIS_SUM].

In order to use the functions, the following strategy has to be followed:

- The orbit initialisation is requiered if the input segments are given in relative orbits. Normally, if the time segments come from visibility functions, the *xo_orbit_id* structure will be already initialised.
- Call the requiered function for segment manipulation.
- The output time segments are returned as dynamical arrays, so when they are not going to be used any more, the arrays should be freed.

Example 4.19 - I: Time segments manipulation (Intersection example)

```
/* Variables */
long orbit_type, start_orbit, stop_orbit,
     start_cycle, stop_cycle;
long swath_flag;
char swath_file[256];
char zone_id[9], zone_db_file[XV_MAX_STR];
long projection, zone_num;
double zone_long[10], zone_lat[10], zone_diam;

long num_1;
long *bgn_orbit_1, *bgn_secs_1, *bgn_microsecs_1, *bgn_cycle_1,
     *end_orbit_1, *end_secs_1, *end_microsecs_1, *end_cycle_1;
long coverage_1;

long num_2;
long *bgn_orbit_2, *bgn_secs_2, *bgn_microsecs_2, *bgn_cycle_2,
     *end_orbit_2, *end_secs_2, *end_microsecs_2, *end_cycle_2;
long coverage_2;

long num_out, order_switch;
long *bgn_orbit_res, *bgn_secs_res, *bgn_microsecs_res, *bgn_cycle_res,
     *end_orbit_res, *end_secs_res, *end_microsecs_res, *end_cycle_res;
long order_criteria;
long   xv_ierr[XV_ERR_VECTOR_MAX_LENGTH];

[...]
```

Variable declaration

```
/* Time and orbit initialisation */
[...]
```

Time and orbit initialisation

```
    /* Getting visibility segments for zone 1 */
    orbit_type = XV_ORBIT_ABS;
    start_orbit = 2900;
    stop_orbit = 2950;
    strcpy(swath_file, "./RA_2_SDF_.N1"); /* SDF */
    strcpy(zone_id, "ZONE_1__");
    strcpy (zone_db_file, "./ZONE_FILE.EEF");

    projection = 0;
    zone_num  = 0;    /* To be able to introduce the zone identifications */
    min_duration = 0.0;

    status = xv_zone_vis_time(&orbit_id, &orbit_type,
                              &start_orbit, &start_cycle,
                              &stop_orbit, &stop_cycle,
                              &swath_flag, swath_file,
                              zone_id, zone_db_file,
                              &projection, &zone_num,
                              zone_long, zone_lat, &zone_diam,
                              &min_duration,
                              &num_1,
                              &bgn_orbit_1, &bgn_secs_1,
                              &bgn_microsecs_1, &bgn_cycle_1,
                              &end_orbit_1, &end_secs_1,
                              &end_microsecs_1, &end_cycle_1,
                              &coverage, xv_ierr);
    [... Error handling...]

    /* Getting visibility segments for zone 2*/
    strcpy(zone_id, "ZONE_2__");

    status = xv_zone_vis_time(&orbit_id, &orbit_type,
                              &start_orbit, &start_cycle,
                              &stop_orbit, &stop_cycle,
                              &swath_flag, swath_file,
                              zone_id, zone_db_file,
                              &projection, &zone_num,
                              zone_long, zone_lat, &zone_diam,
                              &min_duration,
                              &num_2,
                              &bgn_orbit_2, &bgn_secs_2,
                              &bgn_microsecs_2, &bgn_cycle_2,
                              &end_orbit_2, &end_secs_2,
                              &end_microsecs_2, &end_cycle_2,
                              &coverage, xv_ierr);
    [... Error handling...]
```

Getting visibility segments for two different zones.
Afterwards the intersection between the two sets of segments will be computed

```
/* Getting the intersection */
order_switch = XV_TIME_ORDER; /* flag to indicate that the input segments are
                                 already ordered. It saves computation time */
status=xv_time_segments_and (&orbit_id,
            &orbit_type, &order_switch,
            /* input segments list 1*/
            &num_1,
            bgn_orbit_1, bgn_secs_1,
            bgn_microsecs_1, bgn_cycle_1,
            end_orbit_1, end_secs_1,
            end_microsecs_1, end_cycle_1,
            /* input segments list 2*/
            num_2,
            bgn_orbit_2, bgn_secs_2,
            bgn_microsecs_2, bgn_cycle_2,
            end_orbit_2, end_secs_2,
            end_microsecs_2, end_cycle_2,
            /* output segments list */
            &num_out,
            &bgn_orbit_res, &bgn_secs_res,
            &bgn_microsecs_res, &bgn_cycle_res,
            &end_orbit_res, &end_secs_res,
            &end_microsecs_res, &end_cycle_res,
             xv_ierr);

if (status != XV_OK)
{
   func_id = XV_TIME_SEGMENTS_AND_ID;
   xv_get_msg(&func_id, xv_ierr, &n, msg);
   xv_print_msg(&n, msg);
}

/* print outputs */
printf("Outputs for segment intersection: \n");
printf("   Number of segments: %d\n", num_out);
printf("   Segments: Start (Orbit, seconds, microseconds) --
                Stop (Orbit, seconds, microseconds)\n");

for(i=0; i < num_out; i++)
{
   printf("             (%4d, %4d, %6d) -- (%4d, %4d, %6d)\n",
   bgn_orbit_res[i], bgn_secs_res[i], bgn_microsecs_res[i],
   end_orbit_res[i], end_secs_res[i], end_microsecs_res[i]);
}
```

```
    /* Freeing the memory */
    free(bgn_orbit_res);
    free(bgn_secs_res);
    free(bgn_microsecs_res);
    free(bgn_cycle_res);

    free(end_orbit_res);
    free(end_secs_res);
    free(end_microsecs_res);
    free(end_cycle_res);

    free(bgn_orbit_1);
    free(bgn_secs_1);
    free(bgn_microsecs_1);

    free(end_orbit_1);
    free(end_secs_1);
    free(end_microsecs_1);

    free(bgn_orbit_2);
    free(bgn_secs_2);
    free(bgn_microsecs_2);

    free(end_orbit_2);
    free(end_secs_2);
    free(end_microsecs_2);
```

Free memory

```
    /* Closing orbit and time Ids. */

      [...]
```

Close Ids