Earth Observation Mission CFI Software

QUICK START GUIDE

Code:	EO-MA-DMS-GS-0009
Issue:	4.17
Date:	10/05/2019

	Name	**Function**	**Signature**
Prepared by:	Juan Jose Borrego Bote Carlos Villanueva Rubén Castro	Project Engineer	
Checked by:	José Antonio González Abeytua	Project Manager	
Approved by:	José Antonio González Abeytua	Project Manager	

DEIMOS Space S.L.U.
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:	4000102614/1O/NL/FF/ef	Internal	
		Public	
Contract Issuer:	ESA / ESTEC	Industry	X
		Confidential	

External Distribution		
Name	Organisation	Copies

Electronic handling	
Word Processor:	LibreOffice 5.2.3.3
Archive Code:	P/SUM/DMS/01/026-056
Electronic file name:	eo-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	Second version. In line with version 3.7	13/07/07	
2.0	In line with version 4.0	19/07/09	
4.1	In line, even issue numbering, with version 4.1	07/05/10	
4.2	EOCFI delivery 4.2	31/01/11	
4.3	EOCFI delivery 4.3	06/02/12	
4.4	EOCFI delivery 4.4	05/07/12	
4.5	EOCFI delivery 4.5	01/03/13	
4.6	EOCFI delivery 4.6	03/10/13	
4.7	EOCFI delivery 4.7	28/03/14	
4.8	EOCFI delivery 4.8	29/10/14	
4.9	EOCFI delivery 4.9	23/04/15	
4.10	EOCFI delivery 4.10	29/10/15	
4.11	EOCFI delivery 4.11	15/04/16	
4.12	EOCFI delivery 4.12	03/11/16	
4.13	EOCFI delivery 4.13	05/04/17	
4.14	EOCFI delivery 4.14	16/11/17	
4.15	EOCFI delivery 4.15	20/04/18	
4.16	EOCFI delivery 4.16	09/11/18	
4.17	EOCFI delivery 4.17	10/05/19	

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 ACRONYMS, NOMENCLATURE AND TERMINOLOGY

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
EOCFI	Earth Observation 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

1.2 Nomenclature

CFI	A group of CFI functions, and related software and documentation that will be distributed by ESA to the users as an independent unit
CFI function	A single function within a CFI that can be called by the user
Library	A software library containing all the CFI functions included within a CFI plus the supporting functions used by those CFI functions (transparently to the user)

1.3 Note on Terminology

In order to keep compatibility with legacy CFI libraries, the Earth Observation Mission CFI Software makes use of terms that are linked with missions already or soon in the operational phase like the Earth Explorers.

This may be reflected in the rest of the document when examples of Mission CFI Software usage are proposed or description of Mission Files is given.

2 APPLICABLE AND REFERENCE DOCUMENTS

2.1 Reference Documents

[MCD]	Earth Observation Mission CFI Software. Conventions Document. EO-MA-DMS-GS-0001
[MSC]	Earth Observation Mission CFI Software. Mission Specific Customizations. EO-MA- DMS-GS-0018.
[GEN_SUM]	Earth Observation Mission CFI Software. General Software User Manual. EO-MA- DMS-GS-0002.
[F_H_SUM]	Earth Observation Mission CFI Software. EO_FILE_HANDLING Software User Manual. EO-MA-DMS-GS-0008.
[D_H_SUM]	Earth Observation Mission CFI Software. EO_DATA_HANDLING Software User Manual. EO-MA-DMS-GS-007.
[LIB_SUM]	Earth Observation Mission CFI Software. EO_LIB Software User Manual. EO-MA-DMS-GS-003.
[ORB_SUM]	Earth Observation Mission CFI Software. EO_ORBIT Software User Manual. EO-MA-DMS-GS-004.
[PNT_SUM]	Earth Observation Mission CFI Software. EO_POINTING Software User Manual. EO-MA-DMS-GS-005.
[VIS_SUM]	Earth Observation Mission CFI Software. EO_VISIBILITY Software User Manual. EO-MA-DMS-GS-006.

The latest applicable version of [MCD], [F_H_SUM], [D_H_SUM], [LIB_SUM], [ORBIT_SUM], [POINT_SUM], [VISIB_SUM], [GEN_SUM] is v4.17 and can be found at: http://eop-cfi.esa.int/REPO/PUBLIC/DOCUMENTATION/CFI/EOCFI/BRANCH_4X/

3 INTRODUCTION

3.1 Functions Overview

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

- EO_FILE_HANDLING: functions for reading and writing files in XML format.
- EO_DATA_HANDLING: functions for reading and writing Earth Observation Mission files.
- EO_LIB: functions for time transformations, coordinate transformations and other basic transformations.
- EO_ORBIT: functions for computing orbit information.
- EO_POINTING: functions for pointing calculations.
- EO_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 Observation CFI Software can do and the strategies to follow for the different use cases

4 EARTH OBSERVATION CFI USAGE

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

- Reading XML files
- Writing XML files
- Reading/writing Earth Observation 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 Observation 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 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 characterize 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	EO_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	EO_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	EO_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 had been previously initialised
xp_atmos_id	EO_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	EO_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
xp_sat_nom_trans_id	EO_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 was previously

				initialised
xp_sat_att_trans_id	EO_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 was previously initialised
xp_instr_trans_id	EO_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 had been previously initialised
xp_attitude_id	EO_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 had 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 were previously initialised
xp_target_id	EO_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 had been previously initialised but, it does not necessary require that xp_dem_id and xp_atmos_id _id were 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.
xv_swath_id	EO_VISIBILITY	It stores the swath data used in computations.	Output parameter in the initialisation. Input parameter in swath computation routines, visibility routines	xv_swath_id = xp_atmos_id + swath data. It does not require that xp_atmos_id had been previously initialised.

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 an 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.*

Next figure shows the dependency between the Ids.

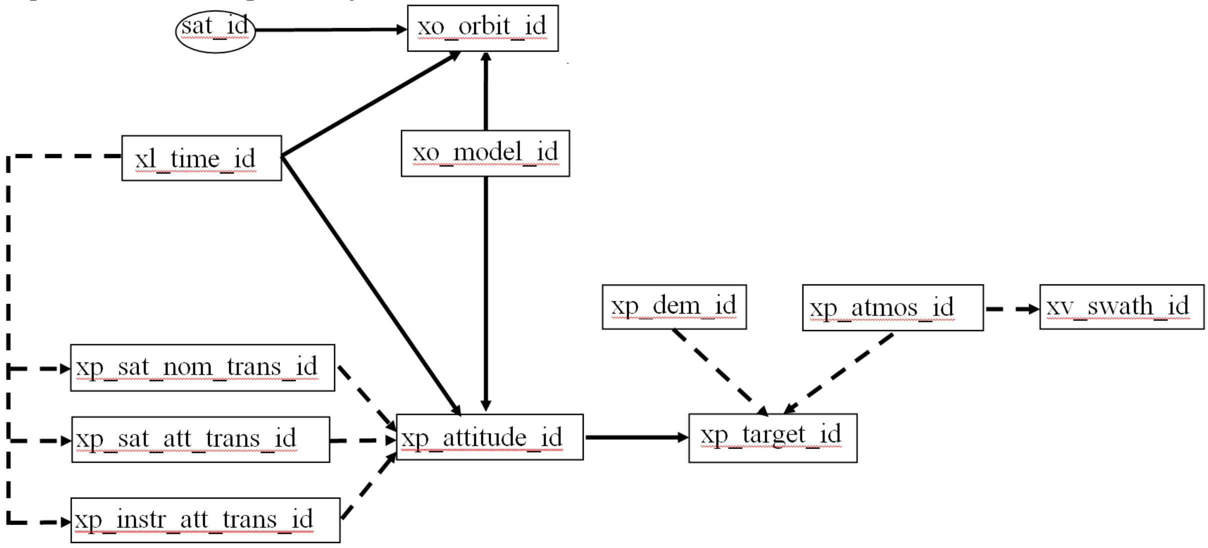

Figure 1: Hierarchical structure of the initialisation variables in the CFI

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 Observation 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 EO_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

Variables 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 EO_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

Variables 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" );
}
```

Writing routines

```
 /* 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" );
 }
```

Write file to disk

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

Close File

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

 [...]
```

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 Observation files

The Earth Observation 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

The versions of these Earth Observation files that are currently supported for reading and writing are described in [D_H_SUM].

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

When reading files, the user should be aware 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.
- The reading functions for each of the file types, does not read the fixed header. The fixed header could be read independently using the CFI function **xd_read_fhr**.
- When reading the fixed header with **xd_read_fhr**, the schema name is not read (the "schema" element in the output structure **xd_fhr** will be set to "_NOSCHEMA_") . If required, the schema name and version should be read independently with the CFI functions in explorer_file_handling.

When writing files, the user should be aware that:

- The schema name and version can be written in the file in the following ways:
  - Setting the schema name in the "schema" element in the **xd_fhr** structure. When calling the **xd_write_*xxx*** function, the schema name and version will be written in the file. Note that if the schema name is set to "_NOSCHEMA_", the schema attributes will no be written in the file.
  - After writing the file, by calling the function **xf_set_schema** (in explorer_file_handling).
- The CFI function **xd_select_schema** allows to get the default schema name with which the file to be written is compliant.

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

Variables declaration

```
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;
```

Open File

```
/* 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);
}
```

Using data structure

```
/* 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);

[...]
```

Writing another OSF with the same data

```
/* 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);
}
```

```
[...]
```

Free data structure

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

4.6 Verifying XML files

Most of Earth Observation 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 executable **xml_validate**. Note that

- The file can be validated using the default schema that is written in the root tag of the file.
- Or it can be validated specifying another schema in the interface of the function.

Note also that is is possible to get the last supported schema name used by the current CFI version by calling the function **xd_select_schema**.

Following there are two examples showing the use of this function. For a detailed explanation about these functions 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 validation

```
/* 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 Configure position/attitude interpolator (decimation)

The list of orbit state vectors and attitude records can be configured according to user need. This can be done in structures corresponding to orbit or attitude files, using the corresponding functions (see [D_H_SUM] for detailed explanation):

- xd_orbit_file_decimate, for orbit files.
- xd_attitude_file_decimate, for attitude files.

These functions decimate the input record list according to input decimate-delta time.

Example 4.7: Configuring orbit file interpolator

Variable declaration & Initialisation

```
/* Variables */
xd_fhr fhr_in, fhr_out;
xd_orbit_file osv_in, osv_out;
double decimation_delta_time;
long status;

[Here read orbit file (osv_in) and fixed header (fhr_in) as
 explained in section 4.5]
```

File validation

```
/* Decimate the file */
decimation_delta_time = 5.; /* seconds */
status = xd_orbit_file_decimate(&fhr_in, &osv_in, decimation_delta_time,
                                &fhr_out, &osv_out, ierr);

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

4.8 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:

- Initialization from a single or multiple orbit files (**xl_time_ref_init_file**).
- Initialization from a structure data containing data read from files or user data (**xl_time_id_init**).
- Initialization 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**).

Next figure represents the data flow for the *xl_time_id* structure.

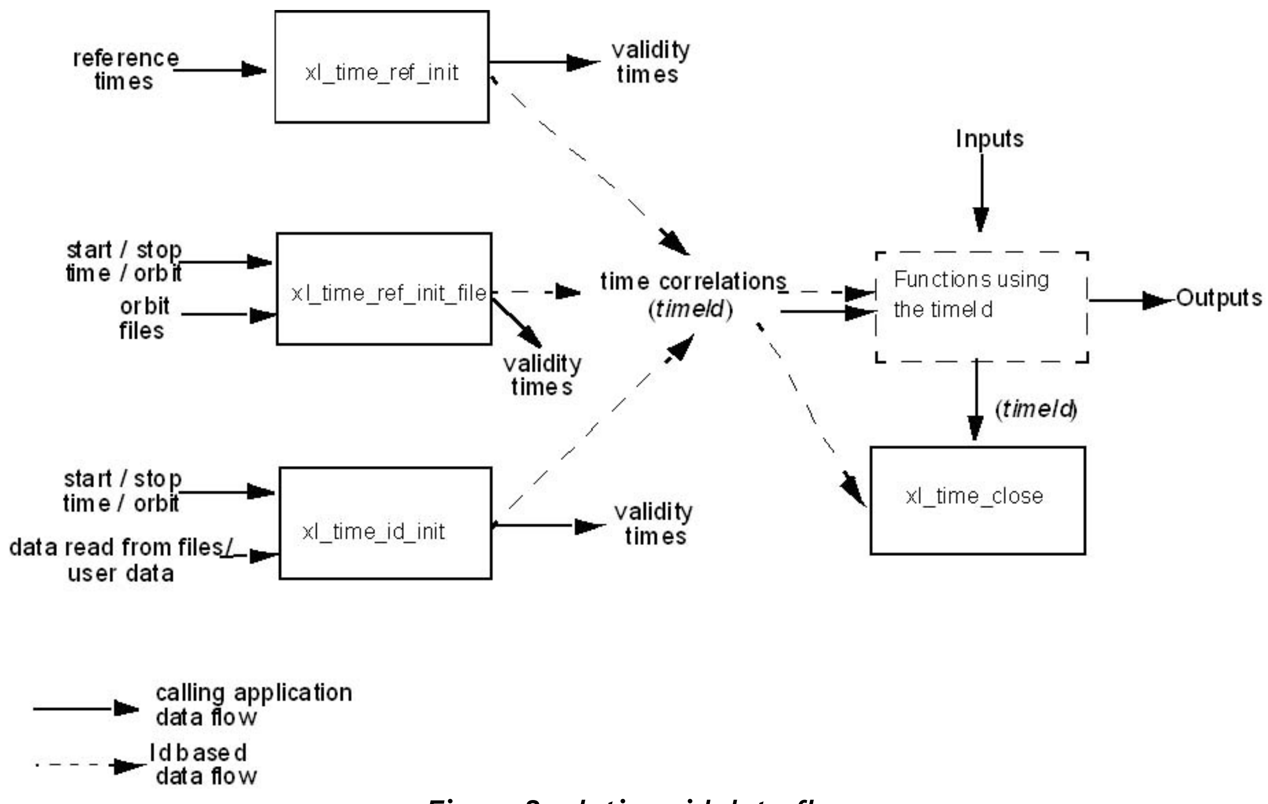

Figure 2: xl_time_id data flow

Examples showing the usage of the time initialization can be found in section 4.9.

4.9 Time transformations

The Earth Observation 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 4.8). 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 4.9 - I: Time transformations. Initialization with an IERS file

Variable declaration

```
/* 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;
```

Time Initialisation

```
/* 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;


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)
```

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

```
{
    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);
}
```

```
/* 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);

/* 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);
```

Time Initialisation

Time Operations

```
    xl_print_msg(&n, msg);
    if (status <= XL_ERR) return(XL_ERR);
}
```

```
[...]
```

Close Time correlation

```
/* 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 4.9 - II: Time transformations. Initialisation with given time correlations.

Variable declaration

```
/* 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;
```

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, 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 Operations

```
/* 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);
}
```

Close Time correlation

Example 4.9 - III: Time transfromation with executable file.

The following command line does the same transformation than the code in :

```
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
```

4.10 Other time calculations

Besides the time transformation functions shown in section , 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 4.10 - 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);
[...]
```

4.11 Using different astronomical models

The EOCFI 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 EOCFI 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 EOCFI 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 EOCFI functions will use the default models.
- The model_id is used as an input parameter in the EOCFI 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 (model_id no initialised)

4.12 Quaternion interpolation/extrapolation

The Earth Observation CFI software provides a function to obtain a quaternion interpoling other 2 quternions: **xl_quaternions_interpol**. If the requested time is between the 2 provided quaternions, the Slerp algorithm is used; if not, an extrapolation is done.

For a detailed description refer to [LIB_SUM].

Example 4.12 - I: Quaternion interpolation/extrapolation

Variable declaration

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

double quaternion1[4], quaternion2[4], quaternion_out[4];
xl_quaternions_interpol_cfg quaternions_interpol_cfg;
double time1_utc, time2_utc, time_inter
```

Prepare inputs and compute interpolated quaternion

```
quaternion1[0] =  0.20552306629887460;
quaternion1[1] =  0.69459322406608592;
quaternion1[2] = -0.029401009484355579;
quaternion1[3] =  0.68879322219508621;
time1_utc      =  1646.7684374999999;

quaternion2[0] =  0.20531498036069915;
quaternion2[1] =  0.69455593356260259;
quaternion2[2] = -0.029187997208036852;
quaternion2[3] =  0.68890193410343314;
time2_utc      = 1646.7684444444444;

time_inter = time1_utc + (time2_utc - time1_utc) *0.25;

/* Call xl_quaternions_interpol function */
status = xl_quaternions_interpol(&quaternions_interpol_cfg, &time1_utc,
                                 quaternion1, &time2_utc, quaternion2,
                                 &time_inter, quaternion_out, xl_ierr);

/* Error handling */
if (status != XL_OK)
{
     func_id = XL_QUATERNIONS_INTERPOL_ID;
     xl_get_msg(&func_id, &status, &n, msg);
     xl_print_msg(&n, msg);
}
```

4.13 Coordinate transformations

The Earth Observation 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 to know more about how to initialise the time correlations). The other functions do not need any special action before calling them.

Example 4.13 - 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

```
    /* 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);
    }
```

Change Coordinate System

```
    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]);
    [...]
```

Get geodetics coordinates

```
    /* 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);
```

```
/* 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... ]
```

4.14 Orbit initialization

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 propagation and interpolation

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

- Time correlation initialization (see section 4.8): the *xl_time_id* is needed for the orbital initialisation in the next step.
- Orbital initialization (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[_precise]**).
  - Providing orbit files through the function **xo_orbit_init_file[_precise]**: 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*.
  - Providig data structures containg user data or data read from files through the function **xo_orbit_id_init**. This function is completely equivalent to **xo_orbit_init_file**.
- 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].

Next figure shows the data flow for the orbital calculations.

The sections 4.15 and 4.15 contain examples showing the orbit initialization usage.

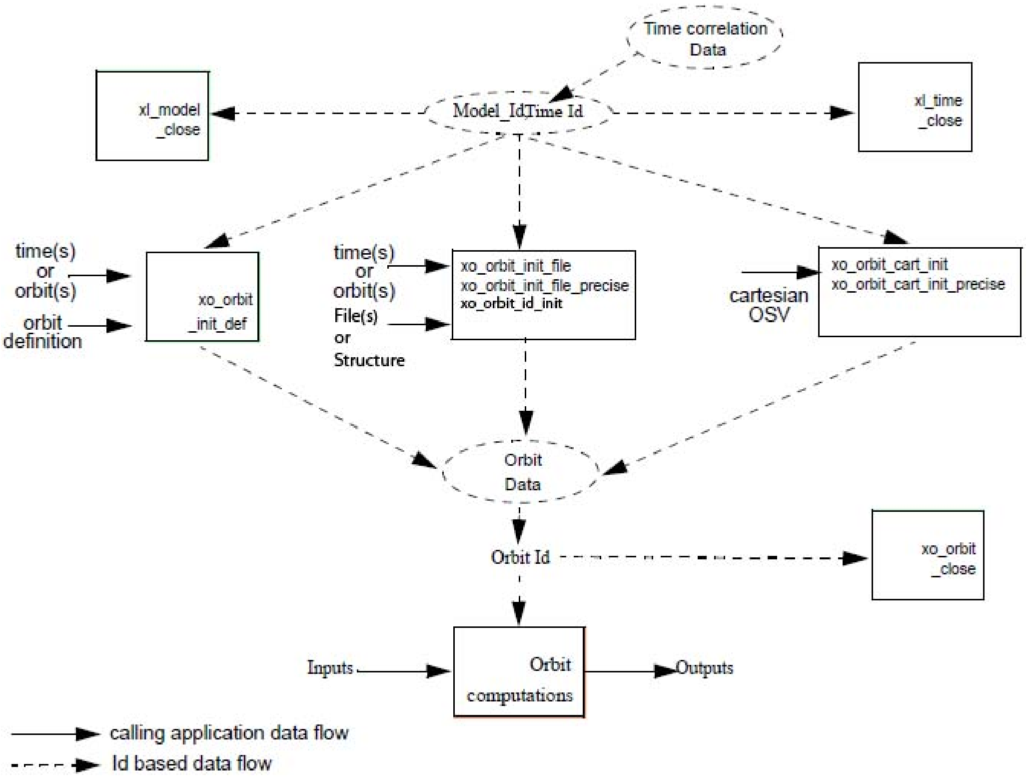

Figure 3: Orbit Information Routines Data Flow

4.15 Orbit calculations

The Earth Observation 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.
- Time, position and velocity vectors in Earth-Fixed associated to a given position on orbit (function **xo_position_on_orbit_to_time**). This position on orbit is defined as the angle between the satellite position and the intersection of the orbital plane with a reference plane (the reference plane is the equator in GM2000, ToD or EF CS).

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

All this functions require the orbit initialisation (section 4.14). 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 4.15 - 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);
}
```

Orbit functions

```
/* 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 4.15 - 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 */
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;
```

Time Initialisation

```
    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);
}
```

Orbit functions

```
/* 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 fromxo_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 close

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

Time close

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

Example 4.15 – III: Orbital calculations with xo_orbit_id_init

Variable declaration

```
/* Variables */
long   status, func_id, n;
char   msg[XL_MAX_COD][XL_MAX_STR];
long   xd_ierr[XD_ERR_VECTOR_MAX_LENGTH];
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};

char orbit_file[XD_MAX_STR];
long extend_osv_flag, reading_osv_flag, time_init_mode;
double range0, range1;
xd_orbit_file orbit_data;

xl_time_id_init_data  time_init_data;
xd_eocfi_file         eocfi_file_array[1];
long time_model, time_init_mode, time_ref;
double time0, time1;
long orbit0, orbit1;
double val_time0, val_time1;

xo_orbit_id_init_data orbit_init_data;
long orbit_file_mode;

long abs_orbit;
double result_vector[XO_ORBIT_INFO_EXTRA_NUM_ELEMENTS];
```

Read orbit data for time and orbit initialization

```
 /* Time initialisation */
strcpy (orbit_file, "../data/EARTH_EXPLORER_FRO");
extend_osv_flag  = XL_FALSE;
reading_osv_flag = XL_TRUE;
time_init_mode   = XL_SEL_FILE;
time_ref         = XL_TIME_UTC;

status = xd_read_orbit_file (orbit_file,   &extend_osv_flag,
                             &time_init_mode,  &time_ref,
                             &range0, &range1, &reading_osv_flag,
```

```
                                            &orbit_data,
                                            xd_ierr);
    if (status != XD_OK)
    {
         func_id = XD_READ_ORBIT_FILE_ID;
         xd_get_msg(&func_id, xd_ierr, &n, msg);
         xd_print_msg(&n, msg);
    }
```

```
    /* Initialize time_id with xl_time_id_init function */
    eocfi_file_array[0].file_type                = XD_ORBIT_FILE;
    eocfi_file_array[0].eocfi_file.orbit_file = orbit_data;
    time_init_data.data_type = XL_FILE_DATA;
    time_init_data.time_id_init_data.file_set.num_files = 1;
    time_init_data.time_id_init_data.file_set.eocfi_file_array =
                             &eocfi_file_array;
    time_ref = XL_TIME_UTC;
    time_model = XL_TIMEMOD_FOS_RESTITUTED;
    time_init_mode = XL_SEL_FILE;

    status = xl_time_id_init (&time_model, &time_id_init_data,
                              &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_ID_INIT_ID;
        xl_get_msg(&func_id, ierr, &n, msg);
        xl_print_msg(&n, msg);
    }
```

Time id initialization

```
    /* Orbit initialisation: xo_orbit_id_init */
    time_mode = XO_SEL_FILE;
    orbit_mode = XO_ORBIT_INIT_ROF_MODE;
    time_ref = XO_TIME_UTC;
    orbit_init_data.data_type = XL_FILE_DATA;
    orbit_init_data.orbit_id_init_data.file_set.num_files = 1;
    orbit_init_data.orbit_id_init_data.file_set.eocfi_file_array = eocfi_file_array;

    status = xo_orbit_id_init(&sat_id, &model_id, &time_id,
                              &orbit_file_mode, &orbit_id_init_data,
                              &time_init_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_ID_INIT_ID;
        xo_get_msg(&func_id, ierr, &n, msg);
        xo_print_msg(&n, msg);
    }
```

Orbit id initialization

```
    /* Get orbit info */
    abs_orbit = 212;
    status = xo_orbit_info (&orbit_id, &abs_orbit, result_vector, xo_ierr);
    if (status != XO_OK)
```

Get orbit information

```
{
     func_id = XO_ORBIT_INFO_ID;
     xo_get_msg(&func_id, xo_ierr, &n, msg);
     xo_print_msg(&n, msg);
}

/* Free orbit file data memory */
xd_free_orbit_file(&orbit_data);

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

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

Free memory and close ids

4.16 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 4.8)
- Orbit initialisation with any of the initialization routines for orbit (section 4.14).
- 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_validity**.
- Optionally, to obtain ancillary results the user might call the **xo_osv_compute_extra function**.
- Optionally, it can be checked to check if the orbit state vector is compatible with the nominal orbit of a given satellite using the function **xo_osv_check**.

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

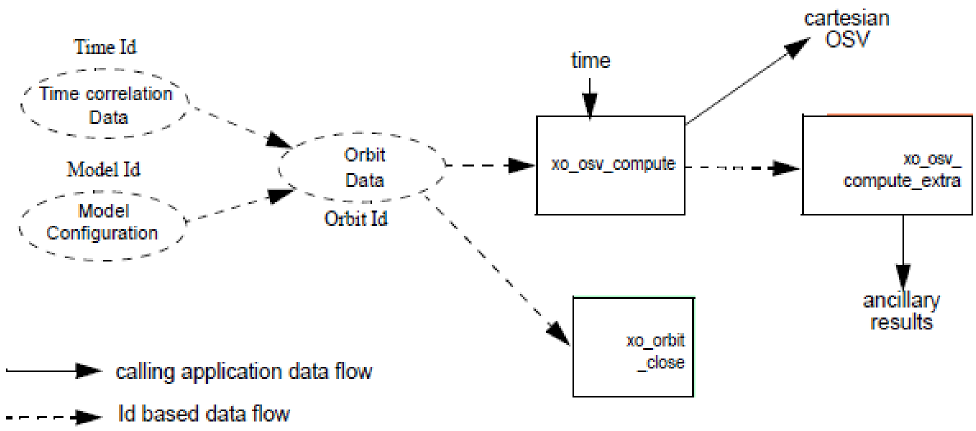

Figure 4: Propag Routines Data Flow

All the previous function are described in [ORB_SUM].

Example 4.16 - 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

4.17 Generation of Earth Observation Orbit Mission Files

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

- Orbit Scenario files: **xo_gen_osf_create_2**.
- 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**.
- TLE files: **xo_gen_tle**

The strategy to follow in all cases is similar:

- Initialise the time correlations (see section 4.8) 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_2**, **xo_gen_osf_repeath_cycle_2**, **xo_gen_osf_add_drift_cycle**.
- Close time correlations (see section 4.8). This step is not needed for the generation of orbit event files.

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

More information can be found in [ORB_SUM].

Next figure shows the calling sequence for the file generation functions.

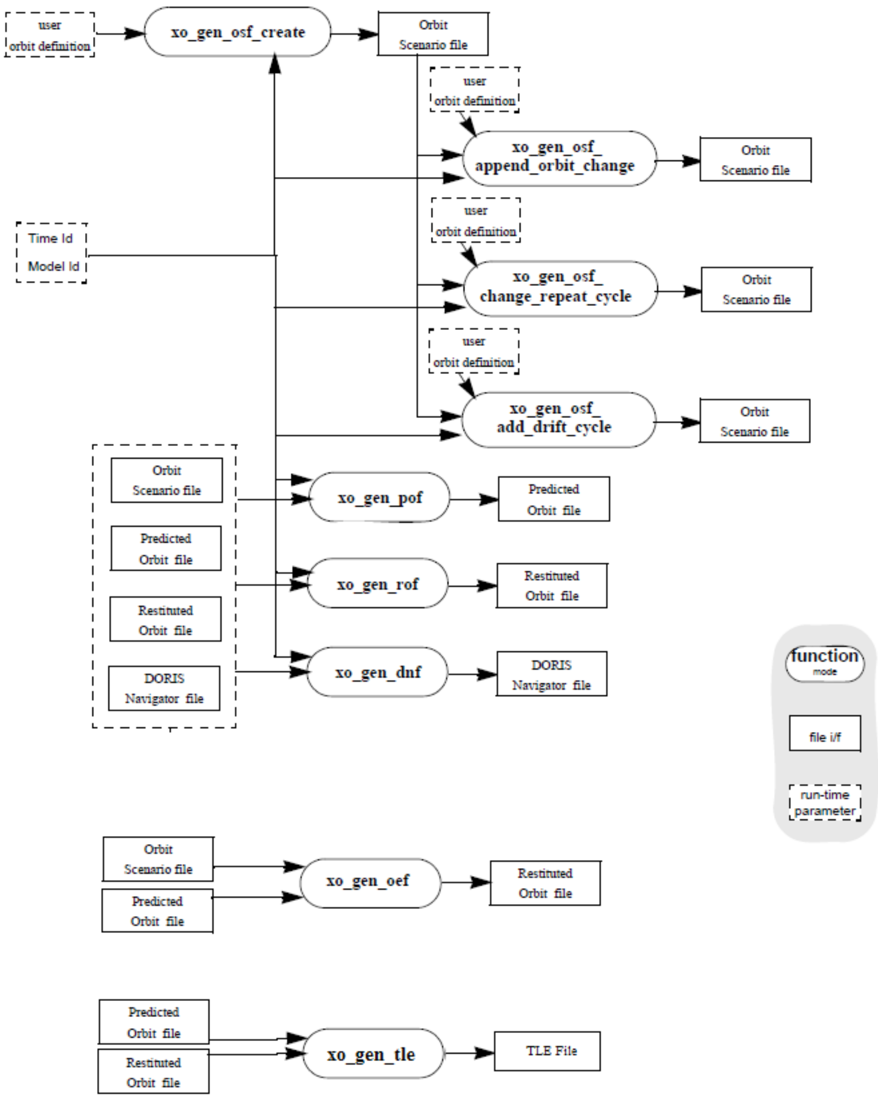

Figure 5: File Generation Calling Sequence

Example 4.17 - I: Orbit Scenario 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_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;

xo_mission_info mission_info;
double date;
xo_ref_orbit_info ref_orbit_info;
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 */
```

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 OSF */

date = 1643.39513888889; /* UTC=2004-07-01_09:29:00.000000 */
mission_info.abs_orbit = 1;
mission_info.rel_orbit = 1;
mission_info.cycle_num = 1;
mission_info.phase_num = 1;
ref_orbit_info.drift_mode = XO_NOSUNSYNC_DRIFT;
ref_orbit_info.rep_cycle = 369;
ref_orbit_info.cycle_len = 5344;
ref_orbit_info.ANX_long = 37.684960;
ref_orbit_info.mlst = 12.0;
ref_orbit_info.mlst_drift = -179.208556;
ref_orbit_info.mlst_nonlinear_drift.linear_approx_validity = 99999
ref_orbit_info.mlst_nonlinear_drift.quadratic_term = 0.
ref_orbit_info.mlst_nonlinear_drift.nof_harmonics = 0
ref_orbit_info.mlst_nonlinear_drift.mlst_harmonics = NULL;
osf_version = 1;

status = xo_gen_osf_create_2(&sat_id, &model_id, &time_id, &date,
                             &mission_info, &ref_orbit_info,
                             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;

/* small change wrt to nominal to check tolerances */
ref_orbit_info.ANX_long = 37.68497;
ref_orbit_info.mlst = mlst
          + mlst_drift*(5345-1)*old_nodal_period/(3600.0*86400.0) + 24.0;

status = xo_gen_osf_append_orbit_change_2(&sat_id, &model_id, &time_id,
                                          output_file_1, &abs_orbit,
                                          &ref_orbit_info, &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);
}
```

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

Time close

Example 4.17 - II: Predicted 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];
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";
```

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 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 4.17 - 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 Restituted Orbit File

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

Time close

Example 4.17 - 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 4.17 - V: Restituted Orbit file generation

Variable declaration

```
/* 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";
```

OEF generation

```
/* 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);
}
```

4.18 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 4.18.1 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 4.18.2 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 4.18.3 for further details about DEM initialisation)

4.18.1 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 6, Figure 7 and Figure 8).

  The three attitudes can be also initialized at the same time with a configuration file using the function **xp_attitude_define**.
- 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 9.

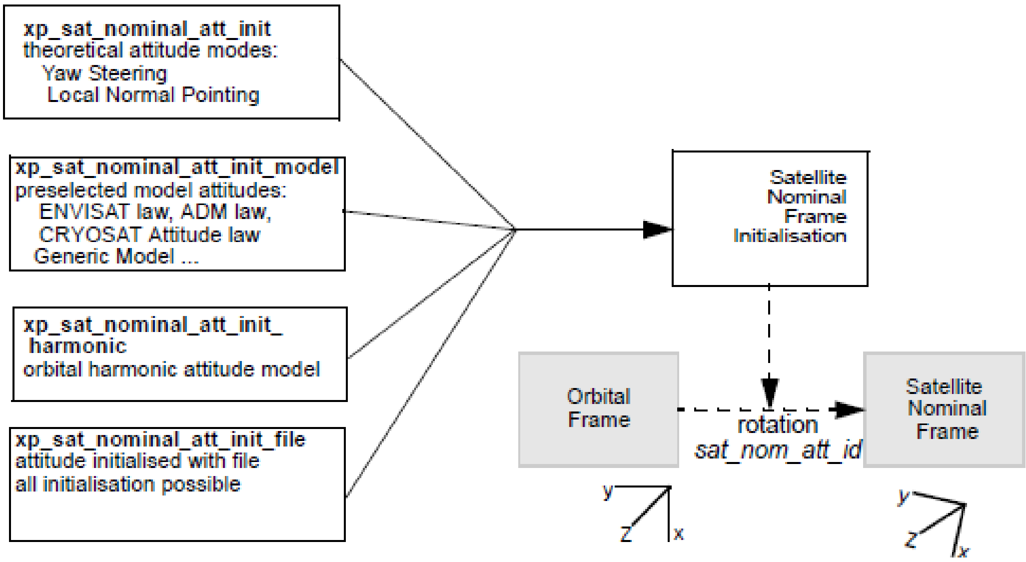

Figure 6: Satellite Nominal Initialisation

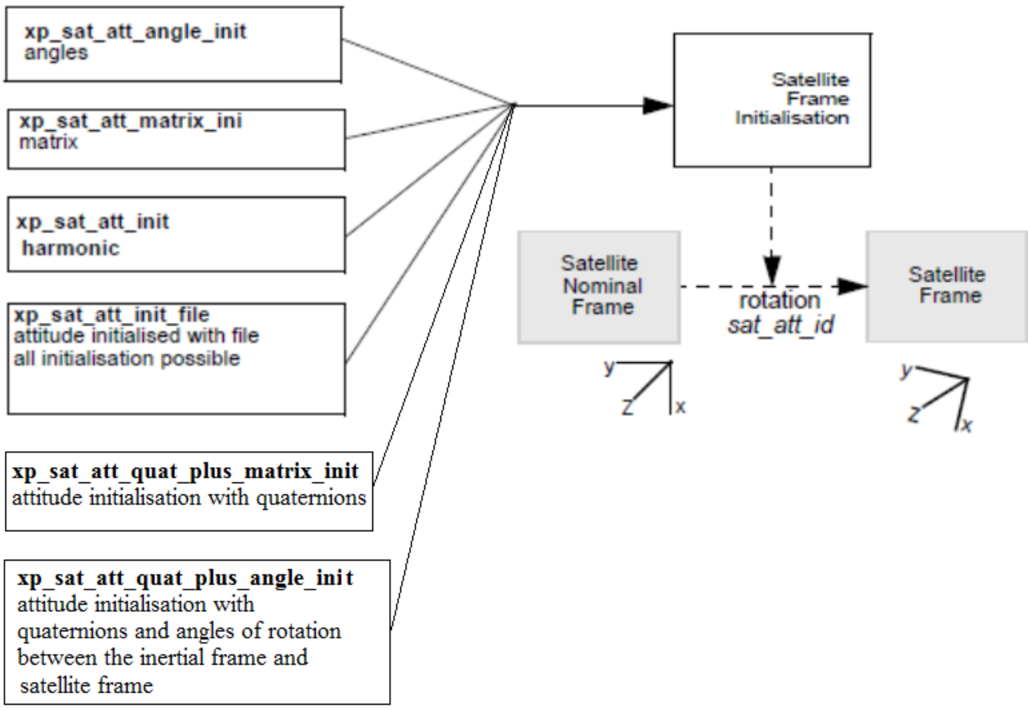

Figure 7: Satellite Initialisation

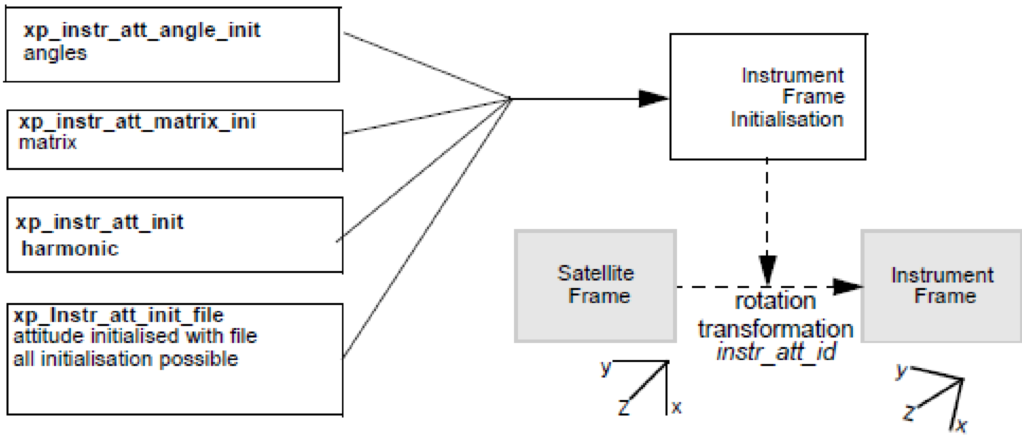

Figure 8: Instrument Initialisation

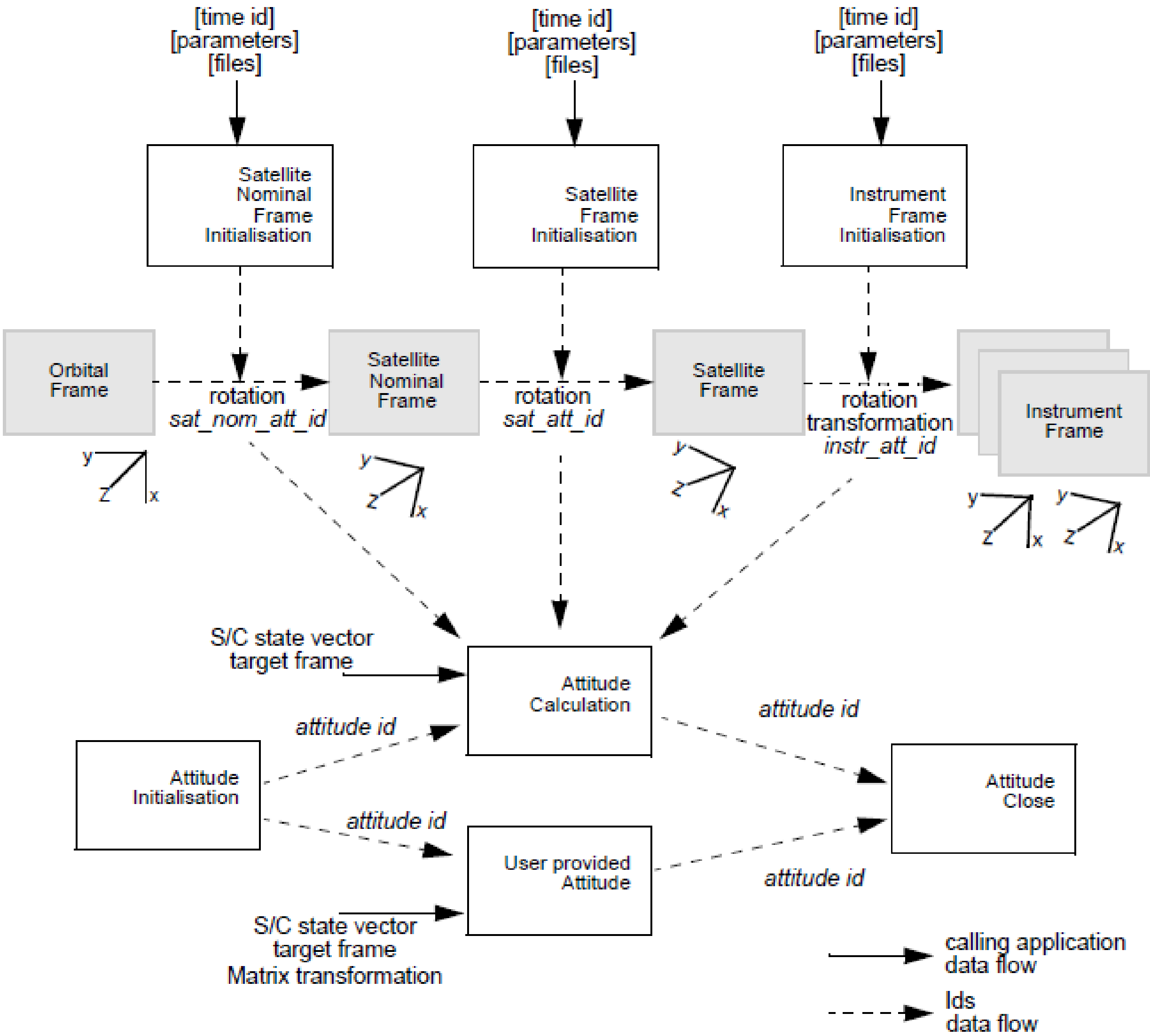

Figure 9: Attitude data flow

Example 4.18 - 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 Initialisations

```
/* 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);
}
```

Time close

Example 4.18 - 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 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);
}
```

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

```
/* 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);
}
```

Attitude close

```
/* 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);
}
```

Attitude frame close

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

4.18.2 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 4.18.4).

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**.

4.18.3 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 4.18.4).

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**.

4.18.4 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_(list)_extra_xxx**.

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

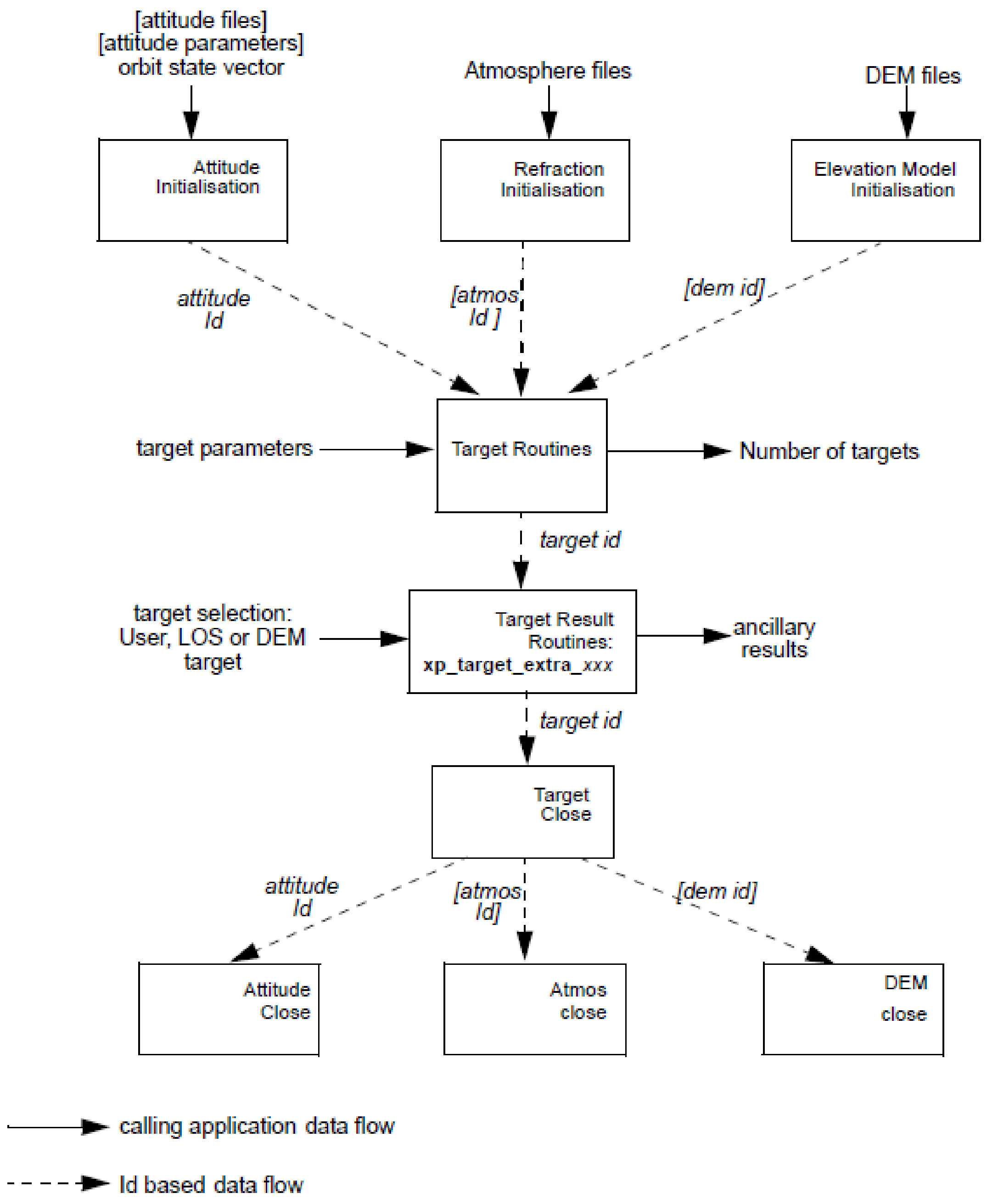

Figure 10: Target data flow

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

Example 4.18 - III: Target Star

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

Satellite Nominal Attitude 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 computations

```
/* 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);
}
```

Computing the target

```
/* 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. Att.

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

Example 4.18 - 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};

[...]
```

Variable declaration

```
/* 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);
}
```

Time 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);
}
```

Orbit Initialisation

```
/* 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 Nominal Attitude Initialisation

```
/* 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 initialisation */

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 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 target data

```
/* Closing Ids */
   status = xp_target_close(&target_id, xp_ierr);
   [...]
```

Closing target

```
} /* end for "i_loop" (End loop for different times )*/
```

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

Close 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);
 [...]
```

Close satellite attitude frame

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

Close DEM

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

Close orbit

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

Close time

```
}

/* end */
```

Example 4.18 – V: Target computation along one orbit using xp_attitude_define, xp_target_list_inter, xp_target_list_extra_vector

```
   [...]
   /* 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};
```

Declare variables

```
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};

char attitude_definition_file[XD_MAX_STR];
xd_attitude_definition_data att_def_file_data;
xp_attitude_def att_def;

xp_instrument_data i_data;
xp_target_output target_num;
xp_target_extra_vector_results_list vector_list;
[...]
```

```
/* Time initialization */
time_model     = XL_TIMEMOD_FOS_PREDICTED;
n_files        = 1;
time_init_mode = XL_SEL_FILE;
time_ref       = XL_TIME_UTC;
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, xl_ierr, &n, msg);
    xl_print_msg(&n, msg);
    if (status <= XL_ERR) return(XL_ERR);
}

/* 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,
                            &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);
 }
```

Initialize time id and orbit id

```
/* Satellite Nominal / Satellite / Instrument attitude frame initialisation
    using the atttitude definition file */
att_def.type = XP_SAT_NOMINAL_ATT;
att_def.sat_nom_trans_id.ee_id = NULL;
att_def.sat_trans_id.ee_id     = NULL;
att_def.instr_trans_id.ee_id   = NULL;

/* Read attitude definition file */
strcpy(attitude_definition_file, "../data/ATT_DEF_AOCS.XML");
status = xd_read_att_def(attitude_definition_file,
                         &att_def_file_data,
                         xd_ierr);
if (status != XD_OK)
{
   func_id = XD_READ_ATT_DEF_ID;
   xd_get_msg(&func_id, xd_ierr, &n, msg);
   xd_print_msg(&n, msg);
}

/* Call xp_attitude_define function */
status = xp_attitude_define(&att_def_file_data,
                            &(att_def.sat_nom_trans_id),
                            &(att_def.sat_trans_id),
                            &(att_def.instr_trans_id),
                            xp_ierr);
if (status != XP_OK)
{
    func_id = XP_ATTITUDE_DEFINE_ID;
    xp_get_msg(&func_id, xp_ierr, &n, msg);
    xp_print_msg(&n, msg);
}

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

Initialize attitudes

```
/* 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)
{[...] }
```

Initialize dem id

```
/* Define a strip of pixels for target_list functions */
i_data.type = XP_AZ_EL_STRIP;
i_data.azimuth_elevation_input_union.azimuth_elevation_strip.azimuth = 270.;
i_data.azimuth_elevation_input_union.azimuth_elevation_strip.min_elevation=50.;
i_data.azimuth_elevation_input_union.azimuth_elevation_strip.max_elevation=70.;
i_data.azimuth_elevation_input_union.azimuth_elevation_strip.step_elevation=10.;
i_data.signal_frequency = 1.e+10;

/* propagate along one orbit */
user_time_start = 2831.00690124781;
user_time_stop = 2831.07112143130;

time_step = 500/86400.0;
```

Prepare inputs for target list computations

```
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, &time,
                           pos, vel, acc, xo_ierr);
   if (status != XP_OK)
   {
     [...]
   }


/* Compute Attitude using the calculated state vector */
target_frame = XP_SAT_NOMINAL_ATT;
status = xp_attitude_compute(&model_id, &time_id,
                             &att_def.sat_nom_trans_id,
                             &att_def.sat_trans_id,
                             &att_def.instr_trans_id,
                             &attitude_id,
                             &time_ref, &time,
                             pos, vel, acc,
                             &target_frame,
                             xp_ierr);
if (status != XP_OK)
{
    [...]
}

/* Get the intersection target */
sat_id       = XP_SAT_ENVISAT;
inter_flag   = XP_INTER_1ST;
deriv        = XP_NO_DER;
geod_alt     = 0.;
```

Compute state vector and attitude for current time

```
/* Call xp_target_list_inter function */
status = xp_target_list_inter(&sat_id, &attitude_id, &atmos_no_ref_id,
                              &dem_id, &deriv, &inter_flag, &inst_data,
                              &geod_alt,
                              /* output */
                              &target_num,
                              &target_id,
                              xp_ierr);
if (status != XP_OK)
{
    func_id = XP_TARGET_LIST_INTER_ID;
    xp_get_msg(&func_id, xp_ierr, &n, msg);
    xp_print_msg(&n, msg);
}

/* Get User, LOS and DEM Targets Data */
choice = XP_NO_DER;
target_type = XP_USER_TARGET_TYPE;

/* Get target parameters; call xp_target_list_extra_vector function */
status = xp_target_list_extra_vector(&target_id,
                                     &choice, &target_type,
                                     &vector_list, xp_ierr);
```

Compute targets and extract information

```
if (status != XP_OK)
{
    func_id = XP_TARGET_LIST_EXTRA_VECTOR_ID;
    xp_get_msg(&func_id, xp_ierr, &n, msg);
    xp_print_msg(&n, msg);
 }
else
{
    printf("\n Target list extra results \n");
    printf("- Num user targets: %ld\n", target_num.num_user_target);
    printf("- First target Position: [%12.3lf,%12.3lf,%12.3lf]\n",
           vector_list.extra_vector_results[0].vector_results[0],
           vector_list.extra_vector_results[0].vector_results[1],
           vector_list.extra_vector_results[0].vector_results[2]);
    [...]
 }
 /* Free memory */
 free(target_num.num_los_target);

}
```

Print output

4.19 Swath initialization

In order to get swath information and perform visibility computations, some data about the swath must be provided. These data have to be stored in the **xv_swath_id** (see section 4.1) before any related calculation could be done. These calculations where the **xv_swath_id** structure are needed are:

- Swath position computation
- Zone, station visibility computations.
- Zone coverage computations.

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

- Time correlation initialization (see section 4.8): the *xl_time_id* is needed for the orbital initialisation in the next step.
- Orbital initialization (see section 4.14): the *xo_orbit_id* is needed in the computations where *xv_swath_id* is used.
- Swath initialization: In this step, the user provides swath information that will be used in further calculations. The data are stored in the *xv_swath_id* "object". The function used to initialise the swath is **xv_swath_id_init**. The swath id can be initialised providing the following information:
  - Swath Definition files (SDF): they contain information about the swath type and geometry and the satellite attitude.
  - 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]. The full description of the function can be found in [VIS_SUM].
- Swath computations: swath position, visibility computations.
- Close swath initialisation calling **xv_swath_id_close**.
- Close orbital initialisation.
- Close Time initialisation.

4.20 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.

Swath files are mainly useful for the visibility calculations (section 4.21) 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 requires the initialisation of the *xo_orbit_id* (section 4.14) and optionally the *xp_atmos_id* (4.18.2) if the swath has to take into account the ray path refraction by the atmosphere.
- Calculating the swath position for a given time (function **xv_swathpos_compute** described in [VIS_SUM]): This operation requires the initialisation of the *xo_orbit_id* and the *xv_swath_id* (see section 4.19).

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

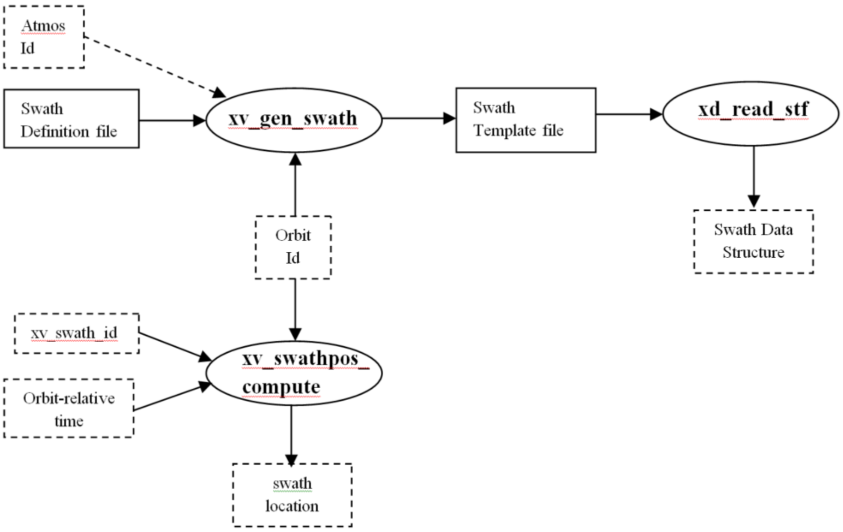

Figure 11: EO_VISIBILITY Data Flow- Swath

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_swathpos_compute**. 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_swathpos_compute** is going to be called for a different orbit, only it is needed if the orbit geometry changes.

Example 4.20 - 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};
xv_swath_id swath_id = {NULL};

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

Declare variables

```
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;

xv_swath_info swath_info;
xv_time swathpos_time;
xv_swath_point_list swath_point_list;
```

Initialize time id and orbit id

```
/* 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...]

/* 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...]
```

Generate swath template file

```
/* 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);
```

```
}
/* Initialize swath id */
swath_info.sdf_file = NULL;
swath_info.stf_file = NULL;
swath_info.nof_regen_orbits = 100;
swath_info.filename = sdf_name;
swath_info.type = XV_FILE_SDF;

status = xv_swath_id_init(&swath_info, &atmos_id,
                          &swath_id, xv_ierr);

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

Initialize swath id

```
/* Call  xv_swathpos_compute function */
swathpos_time.type = XV_ORBIT_TYPE;
swathpos_time.orbit_type = XV_ORBIT_ABS;
swathpos_time.orbit = 2950;
swathpos_time.sec = 100;
swathpos_time.msec = 500000;

status = xv_swathpos_compute(&orbit_id, &swath_id, &swathpos_time,
                             &swath_point_list, xv_ierr);
if (status != XV_OK)
 {
    func_id = XV_SWATHPOS_COMPUTE_ID;
    xv_get_msg(&func_id, xv_ierr, &n, msg);
    xv_print_msg(&n, msg);
 }

/* print outputs */
printf("Outputs: \n");
printf("Swath point (longitude, latitude, altitude): (%lf, %lf, %lf) \n",
       swath_point_list.swath_point[0].lon,
       swath_point_list.swath_point[0].lat,
       swath_point_list.swath_point[0].alt);
```

Compute swath position

```
/* Close swath id */
status = xv_swath_id_close(&swath_id, xv_ierr);
if (status != XV_OK)
{
   func_id = XV_SWATH_ID_CLOSE_ID;
   xv_get_msg(&func_id, xv_ierr, &n, msg);
   xv_print_msg(&n, msg);
}

/* Close orbit_id and time_id*/
[...]
```

Close ids

4.21 Visibility calculations

The Earth Observation 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
- another satellite.
- a star

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

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

- orbital information provided via an orbit Id (see section 4.14).
- requested orbit interval in which the visibilities are to be computed.
- Swath information. The way the swath information is provided depends on the function:
    - For zone (**xv_zonevistime_compute**) and ground station (**xv_stationvistime_compute**) functions: using xv_swath_id.
    - For star (**xv_star_vis_time**) function: using a file with Swath definition or Swath template data.
    - For satellite (**xv_sc_vis_time**) function: no swath information required.
- Information about the target: zone, station, satellite or the star.

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

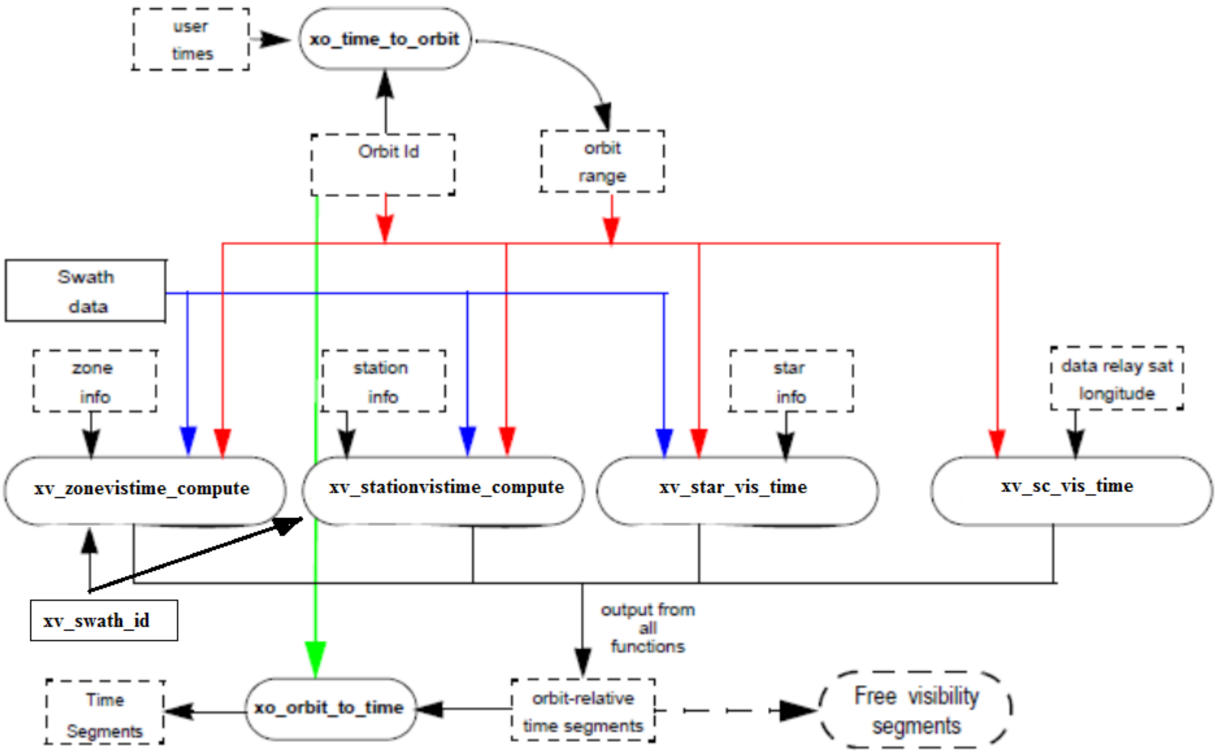

Figure 12: EO_VISIBILITY Data Flow - Visibility

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

For those functions that require swath file as input (**xv_star_vis_time**), 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. There 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 4.21 - I: Getting visibility segments for a zone

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   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;
xv_zone_info zone_info;
xv_zone_info_list zone_info_list;
xp_attitude_def att_def;
xv_time_interval search_interval;
xv_zonevisibility_interval_list vis_list;
```

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) */

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...]
```

```
/* 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

```
/* Prepare input interval for xv_zonevistime_compute */
interval.tstart.type     = XV_UTC_TYPE;
interval.tstart.utc_time = val_time0;
interval.tstop.type      = XV_UTC_TYPE;
interval.tstop.utc_time  = val_time0 + 1.;

/* Prepare zone for xv_zonevistime_compute */
strcpy(zone_id, "ZANA____");
strcpy (zone_db_file, "./ZONE_FILE.EEF");

zone_info.projection = XD_RECTANGULAR; // Rectangular projection
zone_info.min_duration = 0.0;
zone_info.type = XV_USE_ZONE_FILE;
zone_info.zone_id = zone_id;
zone_info.zone_db_filename = zone_db_file;

zone_info_list.calc_flag = XV_COMPUTE; // Compute extra information
zone_info_list.num_rec = 1; // Only one zone
zone_info_list.zone_info = &zone_info;
```

Prepare interval and zone for visibility computations

```
/* Prepare swath id for xv_zonevistime_compute */
strcpy(swath_file, "./RA_2_SDF_.N1"); /* SDF */
swath_info.type = XV_FILE_SDF; // Initialize with Swath definition file
swath_info.filename = swath_file;
swath_info.nof_regen_orbits = 10; // Regenerate internal STF every 10 orbits
swath_info.sdf_file = NULL;
swath_info.stf_file = NULL;

status = xv_swath_id_init(&swath_info, &atmos_id,
                          &swath_id, xv_ierr);
if (status != XV_OK)
{
  func_id = XV_SWATH_ID_INIT_ID;
  xv_get_msg(&func_id, xv_ierr, &n, msg);
  xv_print_msg(&n, msg);
}
```

Initialize swath id

```
/* Use the attitude in Swath definition file */
att_def.type = XP_NONE_ATTITUDE;
```

Compute visibility segments

```
/* Calling xv_zonevistime_compute */
status = xv_zonevistime_compute(&orbit_id, &att_def, &swath_id,
                                &zone_info_list, &search_interval,
                                &vis_list, xv_ierr);
if (status != XV_OK)
{
  func_id = XV_ZONEVISTIME_COMPUTE_ID;
  xv_get_msg(&func_id, xv_ierr, &n, msg);
  xv_print_msg(&n, msg);
}

/* print outputs */
      printf("Outputs: \n");
printf("Number of segments: %d\n", vis_list.num_rec);
printf("   Segments: Start (UTC start) -- Stop (UTC time)\n");

for(i=0; i < vis_list.num_rec; i++)
{
      printf("              (%.10lf) -- (%.10lf)\n",
      vis_list.visibility_interval[i].time_interval.tstart.utc_time,
      vis_list.visibility_interval[i].time_interval.tstop.utc_time);
}
```

Close time and free memory

```
/* free memory: The cycle are not allocated as the orbit type
is absolute orbits*/
free(vis_list.visibility_interval);


/* Close swath id */
status = xv_swath_id_close(&swath_id, xv_ierr);
if (status != XV_OK)
{
      func_id = XV_SWATH_ID_CLOSE_ID;
      xv_get_msg(&func_id, xv_ierr, &n, msg);
      xv_print_msg(&n, msg);
}

/* 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);
}


/* 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);
}
```

4.22 Time segments manipulation

The EO_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 complement of a list of time segments (**xv_timesegments_compute_not**).
- Getting the intersection of two lists of time segments (**xv_timesegments_compute_and**).
- Getting the union of two lists of time segments (**xv_timesegments_compute_or**)
- Adding or subtracting time durations at the beginning and end of every time segment within a list (**xv_timesegments_compute_delta**).
- Sorting a list of time segments (**xv_timesegments_compute_sort**).
- Merging all the overlapped segments in a list (**xv_timesegments_compute_merge**).
- Getting a subset of the time segments list, such that this subset covers entirely a zone or line swath (**xv_timesegments_compute_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 required 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 required 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 4.22 - I: Time segments manipulation (Intersection example)

```
/* Variables */
[...]
xv_zonevisibility_interval_list vis_list1, vis_list2, vis_list_out;
long xv_ierr[XV_ERR_VECTOR_MAX_LENGTH];
```

Variable declaration

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

```
/* Getting visibility segments for zone 1 */

[...]
status = xv_zonevistime_compute (&orbit_id,
                                 &attitude_def,
                                 &swath_id,
                                 &zone_info_list,
                                 &search_interval,
                                 &vis_list1,
                                 xv_ierr);

[... Error handling…]

/* Getting visibility segments for zone 1 */
[...]
status = xv_zonevistime_compute (&orbit_id,
                               &attitude_def,
                               &swath_id,
                               &zone_info_list2,
                               &search_interval,
                               &vis_list2,
                               xv_ierr);
[... Error handling...]
```

Compute 2 sets of segments

```
/* Getting the intersection */
order_switch = XV_TIME_ORDER; /* flag to indicate that the input segments are
                                 already ordered. It saves computation time */
/* call xv_timesegments_compute_and */
status = xv_timesegments_compute_and(&orbit_id, &order_switch
                                     &vis_list1, &vis_list2,
                                     &vis_list_out, xv_ierr);
if (status != XV_OK)
{
   func_id = XV_TIMESEGMENTS_COMPUTE_AND_ID;
   xv_get_msg(&func_id, xv_ierr, &n, msg);
   xv_print_msg(&n, msg);
}

/* print outputs */
printf("Outputs: \n");
printf("Number of segments: %d\n", vis_list_out.num_rec);
printf("   Segments: Start (UTC start) -- Stop (UTC time)\n");

for(i=0; i < vis_list_out.num_rec; i++)
{
     printf("            (%.10lf) -- (%.10lf)\n",
     vis_list_out.visibility_interval[i].time_interval.tstart.utc_time,
     vis_list_out.visibility_interval[i].time_interval.tstop.utc_time);
```

Perform logic operation on the 2 sets of segments

```
}
```

```
/* Freeing the memory */
free(vis_list1.visibility_interval);
free(vis_list2.visibility_interval);
free(vis_list_out.visibility_interval);



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

Close Ids and free memory

4.23 Zone coverage computations

The EO_VISIBILITY library provides a function to compute the portion of the input zone that is covered by a swath during a set of input time visibility intervals: **xv_zonevistime_coverage**. This function provides information about:

- The area of the zone (km^2)
- Percentage of the covered zone (total coverage)
- Coverage per interval
- Coverage per number of intervals
- Cumulative coverage

Example 4.23 – I: Zone coverage computations using xv_zonevistime_coverage

Declare variables

```
/* Variables */
long xv_ierr[XV_ERR_VECTOR_MAX_LENGTH];
xv_zonevisibility_interval_list vis_list;
xv_zonevisibility_coverage_in  zone_cov_in;
xv_zonevisibility_coverage_out zone_cov_out;

/* For time id initialization, orbit_id initialization,
   swath_id initialization check examples in previous sections*/
[…]
```

Compute a list of visibility segments

```
/* Call xv_zonevistime_compute function */
status = xv_zonevistime_compute(&orbit_id, &att_def, &swath_id,
                                &zone_list, &interval,
                                &vis_list, xv_ierr);

if (local_status != XV_OK)
{
   func_id = XV_ZONEVISTIME_COMPUTE_ID;
   xv_get_msg(&func_id, xv_ierr, &n, msg);
   xv_print_msg(&n, msg);
}
```

Compute coverage

```
/* Prepare inputs for coverage computaiton */
zone_cov_in.type_coverage = XV_COVERAGE_PERCENTAGE_PRECISION;
zone_cov_in.point_geod_distance = 10.;
zone_cov_in.percent_precision = 75.;
zone_cov_in.orbit_id = &orbit_id;
zone_cov_in.attitude_def = &att_def;
zone_cov_in.swath_id = &swath_id;
zone_cov_in.zone_info = zone_list.zone_info;
zone_cov_in.visibility_interval_list = &vis_list;

/* Call  xv_zonevistime_coverage function */
status = xv_zonevistime_coverage(&zone_cov_in, &zone_cov_out, xv_ierr);

if (local_status != XV_OK)
{
   func_id = XV_ZONEVISTIME_COVERAGE_ID;
   xv_get_msg(&func_id, xv_ierr, &n, msg);
   xv_print_msg(&n, msg);
```

```
}
/* Print output */
printf("Total coverage = %lf\n", zone_cov_out.total_coverage);

/* Free memory */
free(vis_list.visibility_interval);
free(zone_cov_out.coverage_per_interval);
free(zone_cov_out.coverage_by_N_intervals);
free(zone_cov_out.cumulative_coverage);

/* Close ids */
[…]
```

Print output and free memory

4.24 Fully-featured program

The following program is a simple simulator that calculates orbit, attitude and geolocation geometric properties based on a simple instrument (push broom sensor at a fixed azimuth and a certain numbers of pixels). Observation are above a certain area. The files used can be found in following subsections.

Example 4.24 – I: Fully-featured program

```
#include <explorer_visibility.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

int main()
{
  // Error variables
  long status;
  long func_id;                    // Function ID
  long n;                          // Number of error messages
  char msg[XV_MAX_COD][XV_MAX_STR]; // Error messages vector

  // Error arrays
  long xd_ierr[XD_ERR_VECTOR_MAX_LENGTH];
  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 xv_ierr[XV_ERR_VECTOR_MAX_LENGTH];

  // Ids
  xl_model_id         model_id          = {NULL};
  xl_time_id          time_id           = {NULL};
  xo_orbit_id         orbit_id          = {NULL};
  xp_atmos_id         atmos_id          = {NULL};
  xp_dem_id           dem_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      att_id            = {NULL};
  xp_target_id        target_id         = {NULL};
  xp_target_id        target_id_list    = {NULL};
  xv_swath_id         swath_id          = {NULL};

  // Needed input files: Orbit, attitude, swath and zone files
  // It is assumed that they are placed in the directory where the program is run
  char orbit_file[512] = "orbit_file.xml";
  char attitude_definition_file[512] = "attitude_definition_file.xml";
  char swath_definition_file[512] = "swath_definition_file.xml";
  char dem_config_file[512] = "dem_config_file.xml";
  char zone_file[512] = "";

  // Common variables
  long n_files; // Number of files
  char *input_files[2];

  // Variables for model initialization
```

Declare variables

```
long mode;
long models[XL_NUM_MODEL_TYPES_ENUM];

// Variables for time correlations initialization
long init_model;
long init_mode;
long time_ref;
double time0, time1;
long orbit0, orbit1;
double val_time0, val_time1;

// Variables for orbit initializations
long sat_id = XO_SAT_METOP_SG_A1;

// Variables to initialize dem_id
long use_dem_flag = 0; // If 0, DEM will NOT be used. If 1, DEM will be used

// Variables for atmosphere initialization
char atmos_dummy_file[]="";

// Variables to initialize swath_id
xv_swath_info swath_info;

// Variables for zone visibility computations
char zone_id[20];
xv_zone_info zone_info;
xv_zone_info_list zone_info_list;
xp_attitude_def att_def;
xv_time_interval interval;
xd_zone_rec zone_data;
long num_zone_points = 5;
xd_zone_point zone_points[5];
xv_zonevisibility_interval_list vis_list;
double time_step;

// Variables to initialize attitudes
xd_attitude_definition_data att_def_data;

// Variables for state vector computations
long propag_model = 0; // Dummy
double pos[3], vel[3], acc[3];

// Variables for attitude computations
long target_frame;

// Variable for target computations
long nof_pixels = 3;
double azimuth_list[3]   = {90.,90.,270.};
double elevation_list[3] = {75.,90.,60.};
long deriv = XP_NO_DER;
double geod_alt = 0.;
double az_rate = 0., el_rate = 0.;
long iray = 0; // dummy
double freq = 1e10;
long num_user_target, num_los_target;
long choice = XP_TARG_EXTRA_MAIN_GEO;
long target_type = XP_USER_TARGET_TYPE;
long target_number = 0; // First target
double main_results[XP_SIZE_TARGET_RESULT_MAIN];
double main_results_rate[XP_SIZE_TARGET_RESULT_MAIN];
```

Declare variables

```
double main_results_rate_rate[XP_SIZE_TARGET_RESULT_MAIN];
xp_instrument_data inst_data;
xp_target_output target_num;
xp_azimuth_elevation az_el_list[3];
xp_target_extra_main_results_list main_results_list;
long inter_flag = XP_INTER_1ST;

// auxiliary variables
long seg_i, pixel_i, az_i, time_i;
long time_points_in_interval;
double loop_time;

// Output files
char csv_info[] = "out.csv";
FILE *fp = NULL;

// Initialize default model id
mode = XL_MODEL_DEFAULT;
status = xl_model_init( &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);
}

// Initialize time correlations with orbit file
init_model = XL_TIMEMOD_AUTO; // Detect automatically the type of orbit file
n_files = 1;
input_files[0] = orbit_file;
init_mode = XL_SEL_FILE; // Use all file, time and orbit intervals are dummy
time_ref  = XL_TIME_UTC;
time0 = time1 = 0.;
orbit0 = orbit1 = 0;

status = xl_time_ref_init_file(&init_model, &n_files, input_files,
                               &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, xl_ierr, &n, msg);
  xl_print_msg(&n, msg);
}

// Initialize orbit
sat_id = XO_SAT_METOP_SG_A1; // METOP satellite
init_model = XO_ORBIT_INIT_AUTO; // Detect automatically the type of orbit file
n_files = 1;
input_files[0] = orbit_file;
init_mode = XL_SEL_FILE; // Use all file, time and orbit intervals are dummy
time_ref  = XL_TIME_UTC;
time0 = time1 = 0.;
orbit0 = orbit1 = 0;

status =  xo_orbit_init_file(&sat_id, &model_id, &time_id,
                             &init_model, &n_files, input_files,
                             &init_mode, &time_ref,
```

Declare variables

Model, Time Correlation and Orbit Initialization

```
                                    &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);
    }

    // Initialize dem id if required
    if (use_dem_flag)
    {
      init_mode  = XD_DEM_GETASSE30_V1; // it is not needed, but it is better to have it
                                  // in line with the content of DEM configuration file
      init_model = 0; // Dummy parameter
      status = xp_dem_init(&init_mode, &init_model, dem_config_file,
                           &dem_id,
                           xp_ierr);
      if (status != XP_OK)
      {
        func_id = XP_DEM_INIT_ID;
        xp_get_msg(&func_id, xp_ierr, &n, msg);
        xp_print_msg(&n, msg);
      }

    }

    // Initialize atmos id with no atmosphere model
    init_mode = XP_NO_REF_INIT; // No atmosphere model
    init_model = 0; // dummy parameter
    status = xp_atmos_init(&init_mode, &init_model, atmos_dummy_file,
                           &atmos_id, xp_ierr);
    if (status != XP_OK)
    {
      func_id = XP_ATMOS_INIT_ID;
      xp_get_msg(&func_id, xp_ierr, &n, msg);
      xp_print_msg(&n, msg);
    }


    // Initialize swath id with Swath definition file
    swath_info.type = XV_FILE_SDF; // Initialize with Swath definition file
    swath_info.filename = swath_definition_file;
    swath_info.nof_regen_orbits = 10; // Regenerate internal STF every 10 orbits
    swath_info.sdf_file = NULL;
    swath_info.stf_file = NULL;

    status = xv_swath_id_init(&swath_info, &atmos_id,
                              &swath_id, xv_ierr);
    if (status != XV_OK)
    {
      func_id = XV_SWATH_ID_INIT_ID;
      xv_get_msg(&func_id, xv_ierr, &n, msg);
      xv_print_msg(&n, msg);
    }
```

DEM and Atmosphere Initialization

Swath Initialization

```
// Initialize satellite attitudes to perform.
// We use an attitude definition file, this way we can initialize all the
// atttitudes at the same time.
status = xd_read_att_def(attitude_definition_file,
                         &att_def_data, xd_ierr);
if (status != XD_OK)
{
   func_id = XD_READ_ATT_DEF_ID;
   xd_get_msg(&func_id, xd_ierr, &n, msg);
   xd_print_msg(&n, msg);
}

att_def.type = XP_SAT_ATT; // the attitude definition file has satellite nominal
                           // and satellite attitudes
status = xp_attitude_define(&att_def_data,
                            &att_def.sat_nom_trans_id,
                            &att_def.sat_trans_id,
                            &att_def.instr_trans_id,
                            xp_ierr);
if (status != XP_OK)
{
   func_id = XP_ATTITUDE_DEFINE_ID;
   xp_get_msg(&func_id, xp_ierr, &n, msg);
   xp_print_msg(&n, msg);
}

status = xp_attitude_init(&att_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);
}

// Define zone (Spain polygon)
zone_points[0].pt_long =  3.813258057950049;
zone_points[0].pt_lat  = 42.5091355830345;
zone_points[1].pt_long = -1.706802028054565;
zone_points[1].pt_lat  = 35.82814264689165;
zone_points[2].pt_long = -9.155852980327245;
zone_points[2].pt_lat  = 36.05894971586847;
zone_points[3].pt_long = -9.876611106388474;
zone_points[3].pt_lat  = 44.21197011498435;
zone_points[4].pt_long =  3.813258057950049;
zone_points[4].pt_lat  = 42.5091355830345;

strcpy(zone_data.zone_id, "TEST_ZONE"); // Zone defined in zone file
strcpy(zone_data.description, "Spain zone");
strcpy(zone_data.surface, "");
strcpy(zone_data.creator, "User");
zone_data.zone_type  = XD_POLYGON;
zone_data.projection = XD_RECTANGULAR;
zone_data.zone_diam  = 0.;
zone_data.num_points = num_zone_points;
zone_data.zone_point = zone_points;

zone_info.projection = XD_RECTANGULAR; // Rectangular projection
zone_info.min_duration = 0.0;
zone_info.type = XV_USE_ZONE_DATA; // Use zone data
```

Attitude Initialization

Zone Definition

```
zone_info.zone_id = NULL;
zone_info.zone_db_filename = NULL;
zone_info.zone_data = &zone_data;

zone_info_list.calc_flag = XV_COMPUTE; // Compute extra information
zone_info_list.num_rec = 1; // Only one zone
zone_info_list.zone_info = &zone_info;

// Define visibility interval from start of orbit validity and 1 day more.
interval.tstart.type     = XV_UTC_TYPE;
interval.tstart.utc_time = val_time0;
interval.tstop.type      = XV_UTC_TYPE;
interval.tstop.utc_time  = val_time0 + 1.;

status = xv_zonevistime_compute(&orbit_id, &att_def, &swath_id,
                                &zone_info_list, &interval,
                                &vis_list, xv_ierr);
 if (status != XV_OK)
{
   func_id = XV_ZONEVISTIME_COMPUTE_ID;
   xv_get_msg(&func_id, xv_ierr, &n, msg);
   xv_print_msg(&n, msg);
}

printf("Number of visibility segments found: %ld\n", vis_list.num_rec);

// Prepare inputs for xp_target_list_inter
for (az_i = 0 ; az_i < nof_pixels ; az_i ++)
{
  az_el_list[az_i].azimuth        = azimuth_list[az_i];
  az_el_list[az_i].elevation      = elevation_list[az_i];
  az_el_list[az_i].azimuth_rate   = 0.;
  az_el_list[az_i].elevation_rate = 0.;
}
inst_data.type = XP_AZ_EL_LIST;
inst_data.azimuth_elevation_input_union.azimuth_elevation_list.num_rec = nof_pixels;
inst_data.azimuth_elevation_input_union.azimuth_elevation_list.az_el_list=az_el_list;
inst_data.signal_frequency = freq;

// Perform some computations in the visibility intervals
// We will use a time step of 5 seconds:
time_step = 5. / 86400.;

// Select the type of target to be computed
if (use_dem_flag) target_type = XP_DEM_TARGET_TYPE;
else target_type = XP_USER_TARGET_TYPE;

// Open csv info file to store information and write header line
fp = fopen(csv_info, "w");
if (fp == NULL)
{
  printf("Error opening file %s for writing\n", csv_info);
  return(0);
}

fprintf(fp, "%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s\n",
        "UTC_TIME",
        "SAT_POSITION_X","SAT_POSITION_Y","SAT_POSITION_Z",
        "SAT_VELOCITY_X","SAT_VELOCITY_Y","SAT_VELOCITY_Z",
        "TARGET_LON_PX1","TARGET_LAT_PX1",
```

Zone Definition

Compute Zone Visibility

```
        "TARGET_LON_PX2","TARGET_LAT_PX2",
        "TARGET_LON_PX3","TARGET_LAT_PX3");

for (seg_i = 0 ; seg_i < vis_list.num_rec ; seg_i ++)
{

  // Loop to visibility interval
  time_points_in_interval =
    (vis_list.visibility_interval[seg_i].time_interval.tstop.utc_time -
     vis_list.visibility_interval[seg_i].time_interval.tstart.utc_time)
    / time_step;

  printf("\tSegment %ld. Number of 5 seconds intervals = %ld\n",
        seg_i, time_points_in_interval);


  for (time_i = 0 ; time_i < time_points_in_interval ; time_i ++)
  {
    loop_time = vis_list.visibility_interval[seg_i].time_interval.tstart.utc_time
                + time_i * time_step;


    // Compute Orbit State Vector at current time
    status = xo_osv_compute(&orbit_id, &propag_model, &time_ref, &loop_time,
                            pos, vel, acc, xo_ierr);
    if (status != XO_OK)
    {
      func_id = XO_OSV_COMPUTE_ID;
      xo_get_msg(&func_id, xo_ierr, &n, msg);
      xo_print_msg(&n, msg);
    }

    // Compute attitude at current time (satellite attitude)
    target_frame = XP_SAT_ATT;
    status = xp_attitude_compute(&model_id, &time_id,
                                  &att_def.sat_nom_trans_id,
                                  &att_def.sat_trans_id,
                                  &att_def.instr_trans_id,
                                  &att_id,
                                  &time_ref, &loop_time,
                                  pos, vel, acc,
                                  &target_frame,
                                  /* output */
                                  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);
    }
```

Loop for Every Visibility Segment

Loop fordifferent times along the visibility segment

Compute OSV and attitude at every time instant

```
// Compute target METHOD 1: Compute pixels one by one
for (pixel_i = 0 ; pixel_i < nof_pixels ; pixel_i ++)
{
  status = xp_target_inter(&sat_id,
                           &att_id, &atmos_id, &dem_id,
                           &deriv, &inter_flag,
                           &(azimuth_list[pixel_i]),
                           &(elevation_list[pixel_i]),
                           &geod_alt,
                           &az_rate, &el_rate, &iray, &freq,
                           /* output */
                           &num_user_target, &num_los_target,
                           &target_id, xp_ierr);
  if (status != XP_OK)
  {
    func_id = XP_TARGET_INTER_ID;
    xp_get_msg(&func_id, xp_ierr, &n, msg);
    xp_print_msg(&n, msg);
  }

  status = xp_target_extra_main(&target_id,
                                &choice, &target_type, &target_number,
                                /* output */
                                main_results,
                                main_results_rate,
                                main_results_rate_rate,
                                xp_ierr);
  if (status != XP_OK)
  {
    func_id = XP_TARGET_EXTRA_MAIN_ID;
    xp_get_msg(&func_id, xp_ierr, &n, msg);
    xp_print_msg(&n, msg);
  }
} // end loop to pixels
```

Loop for Every pixel.
Compute the target for every pixel

```
// Compute target METHOD 2: Compute all pixels at the same time
status = xp_target_list_inter(&sat_id, &att_id, &atmos_id, &dem_id,
                              &deriv, &inter_flag, &inst_data, &geod_alt,
                              /* output */
                              &target_num, &target_id_list, xp_ierr);
if (status != XP_OK)
{
  func_id = XP_TARGET_LIST_INTER_ID;
  xp_get_msg(&func_id, xp_ierr, &n, msg);
  xp_print_msg(&n, msg);
}

status = xp_target_list_extra_main(&target_id_list,
                                   &choice, &target_type,
                                   /* output */
                                   &main_results_list,
                                   xp_ierr);
if (status != XP_OK)
{
  func_id = XP_TARGET_LIST_EXTRA_MAIN_ID;
  xp_get_msg(&func_id, xp_ierr, &n, msg);
```

Compute the target for every pixel.
(Alternative way)

```
        xp_print_msg(&n, msg);
      }

      // Close targets so that attitude can be recomputed in next time
      status = xp_target_close(&target_id, xp_ierr);
      if (status != XP_OK)
      {
        func_id = XP_TARGET_CLOSE_ID;
        xp_get_msg(&func_id, xp_ierr, &n, msg);
        xp_print_msg(&n, msg);
      }

      status = xp_target_close(&target_id_list, xp_ierr);
      if (status != XP_OK)
      {
        func_id = XP_TARGET_CLOSE_ID;
        xp_get_msg(&func_id, xp_ierr, &n, msg);
        xp_print_msg(&n, msg);
      }

      fprintf(fp,
    "%.11lf,%.6lf,%.6lf,%.6lf,%.6lf,%.6lf,%.6lf,%.3lf,%.3lf,%.3lf,%.3lf,%.3lf,%.3lf\n",
              loop_time,
              pos[0],pos[1],pos[2],
              vel[0],vel[1],vel[2],
              main_results_list.extra_main_results[0].main_results[0],
              main_results_list.extra_main_results[0].main_results[2],
              main_results_list.extra_main_results[1].main_results[0],
              main_results_list.extra_main_results[1].main_results[2],
              main_results_list.extra_main_results[2].main_results[0],
              main_results_list.extra_main_results[2].main_results[2]);

      // Free memory
      free(target_num.num_los_target);
      target_num.num_los_target = NULL;
      free(main_results_list.extra_main_results);
      main_results_list.extra_main_results = NULL;
    } // End loop to interval
  } // end loop to visibility intervals

  // Close output file
  fclose(fp);
  fp = NULL;

  // Close ids
  status = xv_swath_id_close(&swath_id, xv_ierr);
  if (status != XV_OK)
  {
    func_id = XV_SWATH_ID_CLOSE_ID;
    xv_get_msg(&func_id, xv_ierr, &n, msg);
    xv_print_msg(&n, msg);
  }

  status = xp_attitude_close(&att_id, xp_ierr);
  if (status != XP_OK)
  {
    func_id = XP_ATTITUDE_CLOSE_ID;
    xp_get_msg(&func_id, xp_ierr, &n, msg);
    xp_print_msg(&n, msg);
  }
```

Close Targets

Write results in output file

Free Memory (Close Ids)

```
status = xp_sat_nominal_att_close(&att_def.sat_nom_trans_id, xp_ierr);
if (status != XP_OK)
{
  func_id = XP_SAT_NOMINAL_ATT_CLOSE_ID;
  xp_get_msg(&func_id, xp_ierr, &n, msg);
  xp_print_msg(&n, msg);
}

status = xp_sat_att_close(&att_def.sat_trans_id, xp_ierr);
if (status != XP_OK)
{
  func_id = XP_SAT_ATT_CLOSE_ID;
  xp_get_msg(&func_id, xp_ierr, &n, msg);
  xp_print_msg(&n, msg);
}

if (use_dem_flag)
{
  status = xp_dem_close(&dem_id, xp_ierr);
  if (status != XP_OK)
  {
    func_id = XP_DEM_CLOSE_ID;
    xp_get_msg(&func_id, xp_ierr, &n, msg);
    xp_print_msg(&n, msg);
  }
}

status = xp_atmos_close(&atmos_id, xp_ierr);
if (status != XP_OK)
{
  func_id = XP_ATMOS_CLOSE_ID;
  xp_get_msg(&func_id, xp_ierr, &n, msg);
  xp_print_msg(&n, msg);
}

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);
}

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);
}

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);
}
```

Free Memory (Close Ids)

```
  // Free memory
  free(vis_list.visibility_interval);
  vis_list.visibility_interval = NULL;

  return(0);
}
```

4.24.1 Orbit Scenario file (orbit_file.xml)


```
<?xml version="1.0"?>
<Earth_Explorer_File xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://eop-cfi.esa.int/CFI
http://eop-cfi.esa.int/CFI/EE_CFI_SCHEMAS/EO_OPER_MPL_ORBSCT_0203.XSD"
schemaVersion="2.3" xmlns="http://eop-cfi.esa.int/CFI">
 <Earth_Explorer_Header>
  <Fixed_Header>
   <File_Name>MA1_TEST_MPL_ORBSCT_20210331T213001_99999999T999999_0001</File_Name>
   <File_Description>Reference Orbit Scenario File</File_Description>
   <Notes/>
   <Mission>MetOpSGA1</Mission>
   <File_Class>TEST</File_Class>
   <File_Type>MPL_ORBSCT</File_Type>
   <Validity_Period>
    <Validity_Start>UTC=2021-03-31T21:30:01</Validity_Start>
    <Validity_Stop>UTC=9999-99-99T99:99:99</Validity_Stop>
   </Validity_Period>
   <File_Version>0001</File_Version>
   <Source>
    <System/>
    <Creator>EO_ORBIT:xo_gen_osf_create</Creator>
    <Creator_Version>4.7</Creator_Version>
    <Creation_Date>UTC=2015-01-28T15:19:14</Creation_Date>
   </Source>
  </Fixed_Header>
  <Variable_Header>
   <Time_Reference>UT1</Time_Reference>
  </Variable_Header>
 </Earth_Explorer_Header>
 <Data_Block type="xml">
  <List_of_Orbit_Changes count="1">
   <Orbit_Change>
    <Orbit>
     <Absolute_Orbit>1</Absolute_Orbit>
     <Relative_Orbit>1</Relative_Orbit>
     <Cycle_Number>1</Cycle_Number>
     <Phase_Number>1</Phase_Number>
    </Orbit>
    <Cycle>
```

```
    <Repeat_Cycle unit="day">29</Repeat_Cycle>
    <Cycle_Length unit="orbit">412</Cycle_Length>
    <ANX_Longitude unit="deg">0.000000</ANX_Longitude>
    <MLST>21:30:00.000000</MLST>
    <MLST_Drift unit="s/day">0.000000</MLST_Drift>
    <MLST_Nonlinear_Drift>
     <Linear_Approx_Validity unit="orbit">99999</Linear_Approx_Validity>
     <Quadratic_Term unit="s/day^2">0.000000</Quadratic_Term>
     <Harmonics_Terms num="0"/>
    </MLST_Nonlinear_Drift>
   </Cycle>
   <Time_of_ANX>
    <TAI>TAI=2021-03-31T21:30:36.272398</TAI>
    <UTC>UTC=2021-03-31T21:30:01.272398</UTC>
    <UT1>UT1=2021-03-31T21:30:01.272398</UT1>
   </Time_of_ANX>
  </Orbit_Change>
 </List_of_Orbit_Changes>
 </Data_Block>
</Earth_Explorer_File>
```

4.24.2 Attitude Definition File (attitude_definition_file.xml)

```
<?xml version="1.0"?>
<Earth_Explorer_File              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://eop-cfi.esa.int/CFI
http://eop-cfi.esa.int/CFI/EE_CFI_SCHEMAS/EO_OPER_INT_ATTDEF_0203.XSD"
schemaVersion="2.3" xmlns="http://eop-cfi.esa.int/CFI">
 <Earth_Explorer_Header>
  <Fixed_Header>
   <File_Name>S1A_TEST_INT_ATTDEF_00000000T000000_99999999T999999_0004</File_Name>
   <File_Description>Attitude Definition File</File_Description>
   <Notes>
        WARNING: this is an example of file compliant with formats defined and supported
             within the Earth Observation Mission CFI Software.
             This file does not reflect the actual satellite orbit
             attitude or instrument configuration.
             This file shall not be used in production environments.
   </Notes>
   <Mission>Sentinel1A</Mission>
   <File_Class>TEST</File_Class>
   <File_Type>INT_ATTDEF</File_Type>
   <Validity_Period>
    <Validity_Start>UTC=0000-00-00T00:00:00</Validity_Start>
    <Validity_Stop>UTC=9999-99-99T99:99:99</Validity_Stop>
   </Validity_Period>
```

```
    <File_Version>0004</File_Version>
    <Source>
     <System>System Identification as per Ground Segment File Format Standard (PE-TN-ESA-GS-0001)</System>
     <Creator>Creator Identification as per Ground Segment File Format Standard (PE-TN-ESA-GS-0001)</Creator>
     <Creator_Version>Creator Version Identification as per Ground Segment File Format Standard (PE-TN-ESA-GS-0001)</Creator_Version>
     <Creation_Date>UTC=2015-01-01T00:00:00</Creation_Date>
    </Source>
   </Fixed_Header>
   <Variable_Header/>
  </Earth_Explorer_Header>
  <Data_Block type="xml">
   <Attitude_Definition>
    <Sat_Nominal_Att>
     <AOCS_Model>YAW_STEERING_MODE</AOCS_Model>
    </Sat_Nominal_Att>
    <Sat_Att>
     <Angle_Model>
      <Angle_1 unit="deg">1</Angle_1>
      <Angle_2 unit="deg">0</Angle_2>
      <Angle_3 unit="deg">0</Angle_3>
     </Angle_Model>
    </Sat_Att>
    <Instr_Att>
     <None/>
    </Instr_Att>
   </Attitude_Definition>
  </Data_Block>
</Earth_Explorer_File>
```

4.24.3 Swath definition file (swath_definition_file.xml)

```
<?xml version = "1.0"?>
<Earth_Explorer_File xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                                         xsi:schemaLocation="http://eop-cfi.esa.int/CFI http://eop-cfi.esa.int/CFI/EE_CFI_SCHEMAS/EO_OPER_MPL_SW_DEF_0302.XSD"
 xmlns="http://eop-cfi.esa.int/CFI" schemaVersion="3.2">
  <Earth_Explorer_Header>
   <Fixed_Header>
    <File_Name>SDF_3MI</File_Name>
    <File_Description>Swath Definition File</File_Description>
    <Notes>Local Normal Pointing + Yaw Steering Attitude (AOCS mode 2)</Notes>
    <Mission>Metop-SG-A</Mission>
    <File_Class>TEST</File_Class>
    <File_Type>MPL_SW_DEF</File_Type>
    <Validity_Period>
     <Validity_Start>UTC=0000-00-00T00:00:00</Validity_Start>
```

```
    <Validity_Stop>UTC=9999-99-99T99:99:99</Validity_Stop>
   </Validity_Period>
   <File_Version>0001</File_Version>
   <Source>
    <System>Manual</System>
    <Creator>ESA/ESTEC</Creator>
    <Creator_Version>1.0</Creator_Version>
    <Creation_Date>UTC=2015-03-15T12:00:00</Creation_Date>
   </Source>
  </Fixed_Header>
  <Variable_Header></Variable_Header>
 </Earth_Explorer_Header>
 <Data_Block type="xml">
  <Swath>
   <Output_File_Description>3MI</Output_File_Description>
   <Output_File_Type>MPL_SWTREF</Output_File_Type>
   <Swath_Type>open</Swath_Type>
   <Num_Swath_Records>1200</Num_Swath_Records>
   <Refraction>
    <Model>NO_REF</Model>
    <Freq unit="MHz">000000000000</Freq>
   </Refraction>
   <List_of_Swath_Points count="3">
    <Swath_Point>
     <Pointing_Geometry>
      <Azimuth unit="deg">+270.000000</Azimuth>
      <Elevation unit="deg">+039.800000</Elevation>
      <Altitude unit="m">+000000.000</Altitude>
     </Pointing_Geometry>
    </Swath_Point>
    <Swath_Point>
     <Pointing_Geometry>
      <Azimuth unit="deg">+090.000000</Azimuth>
      <Elevation unit="deg">+090.000000</Elevation>
      <Altitude unit="m">+000000.000</Altitude>
     </Pointing_Geometry>
    </Swath_Point>
    <Swath_Point>
     <Pointing_Geometry>
      <Azimuth unit="deg">+090.000000</Azimuth>
      <Elevation unit="deg">+039.800000</Elevation>
      <Altitude unit="m">+000000.000</Altitude>
     </Pointing_Geometry>
    </Swath_Point>
   </List_of_Swath_Points>
    <Sat_Nominal_Att>
     <AOCS_Model>2</AOCS_Model>
```

```xml
    </Sat_Nominal_Att>
    <Sat_Att>
     <Angle_Model>
      <Angle_1 unit="deg">1</Angle_1>
      <Angle_2 unit="deg">0</Angle_2>
      <Angle_3 unit="deg">0</Angle_3>
    </Angle_Model>
    </Sat_Att>
    <Instr_Att>
     <None></None>
    </Instr_Att>
  </Swath>
 </Data_Block>
</Earth_Explorer_File>
```

4.24.4 DEM configuration file (dem_config_file.xml)

```xml
<?xml version="1.0"?>

<Earth_Explorer_File xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://eop-cfi.esa.int/CFI http://eop-cfi.esa.int/CFI/EE_CFI_SCHEMAS/EO_OPER_INT_ATTREF_0101.XSD" schemaVersion="1.1" xmlns="http://eop-cfi.esa.int/CFI">
  <Earth_Explorer_Header>
    <Fixed_Header>
      <File_Name>DEM_CONFIG_TEST_FILE</File_Name>
      <File_Description>DEM Configuration File</File_Description>
      <Notes />
      <Mission></Mission>
      <File_Class>TEST</File_Class>
      <File_Type></File_Type>
      <Validity_Period>
        <Validity_Start>UTC=0000-00-00T00:00:00.000000</Validity_Start>
        <Validity_Stop>UTC=9999-99-99T99:99:99.999999</Validity_Stop>
      </Validity_Period>
      <File_Version>1</File_Version>
      <Source>
        <System>CFI Acceptance</System>
        <Creator></Creator>
        <Creator_Version></Creator_Version>
        <Creation_Date>UTC=2010-04-14T17:25:44</Creation_Date>
      </Source>
    </Fixed_Header>
    <Variable_Header />
  </Earth_Explorer_Header>
  <Data_Block type="xml">
   <DEM>
    <DEM_User_Parameters>
     <Directory>/DEM_DATA/DEM</Directory>
```

```
      <Cache_Type>FIFO_CACHE</Cache_Type>
      <Cache_Max_Size size="MB">4096</Cache_Max_Size>
    </DEM_User_Parameters>
    <DEM_Metadata>
      <Dataset_Model>GETASSE30_V1</Dataset_Model>
      <Description></Description>
    </DEM_Metadata>
  </DEM>
 </Data_Block>
</Earth_Explorer_File>
```