Integration Libraries for the open Simulation Framework

OSFI

DEVELOPER'S MANUAL

Code : OPENSF-DMS-OSFI-DM-013
Issue : 1.12
Date : 06/07/2017

	Name	Function	Signature
Prepared by	Enrique del Pozo Jose Julio Ramos Alberto Monescillo Jose Luis Garcia Rui Mestre Javier Martín	Project Engineers	
Reviewed by	Federico Letterio	Project Manager	
Approved by	Federico Letterio	Project Manager	
Signatures and approvals on original			

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

This page intentionally left blank

Document Information

Contract Data	
Contract Number:	22852/09/NL/FF
Contract Issuer:	ESA/ESTEC

Internal Distribution		
Name	Unit	Copies
Internal Confidentiality Level (DMS-COV-POL05)		
Unclassified ☑	Restricted ☐	Confidential ☐

External Distribution		
Name	Organisation	Copies
Michele Zundo Montserrat Pinõl Sole Maurizio De Bartolomei	ESA/ESTEC	1

Archiving	
Word Processor:	MS Word 2010
File Name:	OPENSF-DMS-OSFI-DM-013-112.doc

Document Status Log

Issue	Change description	Date	Approved
1.0	First issue of this document	15/12/08	
1.1	ANSI C programming language support	27/04/09	
1.2	Quality indicators functionality added	26/05/09	
1.3	Mac Installation added	15/03/10	
1.4	OSFI library for F77, IDL and Matlab. Added sections describing the integration of models in F77, IDL and Matlab.	22/09/10	
1.5	Updated OSFI library for Matlab. ❑ Added configuration for IDL. ❑ Section 4.1 now points to openSF ADD, OSFI section. ❑ Added new section 0 with OSFI additional features.	12/11/2010	
1.6	Updated OSFI library for Python.	02/04/2014	
1.7	Updated after review comments from ESA: implemented RIDS OPENSF_v3.2_RID_03, OPENSF_v3.2_RID_04 and OPENSF_v3.2_RID_05 by updating section 4.2.7.	30/04/2014	
1.8	Updated OSFI library for Java Update of AD/RD Reworded introduction	15/08/2016	
1.9	Updated OSFI available API (added existParameter method)	20/10/2016	
1.10	Updated after review comments from ESA: added Python requirements, added reference documentation (doxygen)	18/11/2016	
1.11	Updated due to support to Python 3.X	16/06/2017	
1.12	Update version of compilers used in openSF	06/07/2017	

Table of Contents

List of Tables

List of Figures

1. INTRODUCTION

The open Simulation Framework (openSF) relies on a well-defined set of interfaces [E2E-ICD] that the participating modules have to adhere to. The OSFI activity addressed the definition and development of a set of software libraries to ease the integration of modules into openSF system by providing a ready-made implementation of these interfaces.

Usage of OSFI libraries are therefore a key component to easily develop modules using openSF as orchestrating framework.

Terminology Note: starting with openSF 3.3 the recommended term to identify the orchestrated software components within an E2E simulation is **Module** instead of **Model**.

The text in this document has been amended accordingly however the name of software functions and variables still reflects the old naming convention.

1.1. Purpose

The objective of this document is to provide a detailed description and a development manual for the set of software libraries (OSFI) that can be used during the development and deployment of the modules within an E2E Mission Performance simulator

The intended readerships for this document are model developers and scientists that are in charge of integrate those models into the openSF.

This document is also useful to software engineers responsible of the testing stage.

1.2. Scope

This document shows a detailed description of the integration libraries and an API that can be used as a reference manual by model developers. It also includes a brief architecture description and some examples of use.

This document contains the following sections:

- An introduction (current section 1) for giving a quick overview of the project;
- A list of related documents to provide a documentary background (section 2)
- An introduction to the integration libraries, installation and linking instructions (section 3)
- A description of the architecture, the process logic and some examples of use. It also includes the coding guidelines (section 4)

1.3. Acronyms and Abbreviations

The acronyms and abbreviations used in this document are the following ones:

Acronym	Description
AD	Architectural Design Applicable Document
ADD	Architectural Design Document
API	Application Programming Interface
AR	Acceptance Review Analysis of Requirements
CFI	Customer Furnished Item
CLP	Command Line Parser
CM	Configuration Management Configuration Manager
CMP	Configuration Management Plan
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
DD	Detailed Design
DMS	DEIMOS Space
DRR	Document Review Record
ECP	Engineering Change Proposal
E-R	Entity Relationship
GUI	Graphical User Interface
HW	Hardware
I/F	Interface
I/O	Input/Output
ICD	Interface Control Document
ITT	Invitation To Tender
KOM	Kick-Off Meeting
MD	Managing Director
MMI	Man-Machine Interface
MoM	Minutes of Meeting
MR	Management Review
NCR	Non-Conformance Report
O/S	Operating System
PA	Product Assurance
PDR	Preliminary Design Review
PM	Progress Meeting Project Manager

Acronym	Description
PMP	Project Management Plan
QA	Quality Assurance
RD	Reference Document
RID	Review Item Discrepancy
SOW	Statement Of Work
SPR	Software Problem Report
SR	Software Requirements
SRD	Software Requirements Document
SRN	Software Release Note
SRS	Software Requirements Specification
STR	Software Test Report
SUM	System User Manual
SVS	Software Validation Specification
SW	Software
TBC	To Be Confirmed
TBD	To Be Defined / Decided
TER	Test Execution Record
TN	Technical Note
TP	Test Plan
TR	Test Report
TS	Technical Specification
UML	Unified Modelling Language
URD	User Requirements Document
V&V	Verification & Validation

2. RELATED DOCUMENTS

2.1. Applicable Documents

The following table specifies the applicable documents compliant with OSFI development.

Table 1: Applicable documents

Reference	Code	Title
[OSF-ICD]	openSF-DMS-ICD-001	OpenSF Interface Control Document Issue 3.0
[E2E-ICD]	PE-ID-ESA-GS-464	ESA Generic E2E ICD Issue 1.2.2

2.2. Reference Documents

The following table specifies the reference documents to be taken into account during module development.

Table 2: Reference documents

Reference	Code	Title
[OSF-SUM]	OPENSF-DMS-SUM-001	OpenSF System User Manual Issue 3.8
[OSF-ADD]	openSF-DMS-ADD-001	OpenSF Architecture Design Document Issue 2.2
[OSF-SRD]	openSF-DMS-SRD-001	OpenSF System Requirements Document Issue 3.2
[OSF-DOC]	http://eop-cfi.esa.int/Repo/PUBLIC/DOCUMENTATION/OPENSF/DOXYGEN/	OSFI Doxygen documentation

2.3. Standards

The following table specifies the standards complied with during project development.

Table 3: Standards

Reference	Code	Title	Issue	Date
[XML]	(www.w3.org/TR/xml11/)	Extensible Markup Language (XML) 1.1	Second Edition	Sep 29 2006
[UML]	www.uml.org/#UML2.0)	Unified Model Language (UML)	2.1	Oct 6 2006
[BNF]	(see also en.wikipedia.org /wiki /Backus-Naur_form)	Algol-60 Reference Manual	5	1979

3. GETTING STARTED

3.1. Introduction

In the frame of concept and feasibility studies for the Earth Observation (EO) activities, mission performance in terms of final data products needs to be predicted by means of so-called end-to-end (E2E) simulators.

A specific mission E2E simulator is able to reproduce all significant processes and steps that impact the mission performance and gets simulated final data products.

The open Simulation Framework (openSF) is a generic simulation framework aimed to cope with these major goals. It provides end-to-end simulation capabilities that allow assessment of the science and engineering goals with respect to the mission requirements and it is available for Linux and OSX.

This openSF framework allows the users to integrate and execute pieces of code, «modules» that form the building blocks of a simulation process.

To integrate an external module into the framework, the module needs to fulfil a series of interface requirements detailed in [OSF-ADD] and [OSF-ICD].

The Open Simulation Framework Integration Libraries (OSFI) will be used to ease the integration of modules into the open Simulation Framework.

The Integration Libraries activity provides the module developer with a set of routines with a well-defined public interface hiding the implementation details. This set of routines is currently available in C++, ANSI C, Fortran 90, Fortran 77, IDL, Matlab, Python and Java.

3.2. Conventions used in this Manual

This chapter lists all the conventions used throughout this Developer's Manual

3.2.1. $OSFI_HOME

All through the contents of this Developer Manual, a "variable" called $OSFI_HOME is exhaustively used as a placeholder. The variable value points to the root folder that contains the openSF installation. Typically, this folder could be similar to this:

/home/*user_name*/openSF

This variable matches with one "environment variable" defined in the openSF system. You can find more information in [OSF-ICD].

3.2.2. Data types

Every requested or given piece of data in OSFI is formatted in one of the following data types:

- *STRING*. A string of alphanumeric characters with a size not greater than 255.

- *INTEGER*. Integer number (no decimal part) between -2^{31} and 2^{31-1}
- *FLOAT*. Decimal number between 2^{-149} and $(2-2^{-23})\ 2^{127}$
- *BOOLEAN*. TRUE of FALSE.
- *FILE*. The absolute (or $OSFI_HOME-relative) path and name of a file into the file system.
- *FOLDER*. The absolute (or $OSFI_HOME-relative) path and name of a folder into the file system.

3.3. Initial Requirements

The OSFI system is prepared to run in a hardware and software platform with the following requirements. These must be fulfilled before installing the distribution.

3.3.1. Hardware requirements

This software is compatible with the following architectures and operating systems:

- *Operating systems*: Linux, OSX
- *Architectures*: x86-64 (also known as AMD64 or Intel 64)

3.3.2. Software requirements

This is the list of suggested compilers for the sources.

Table 4: suggested compilers for sources

Language	Compiler	Licensing	Distribution Site
Fortran 90	Intel Fortran compiler v 9.1	Several options. There is a free edition for the community	http://support.intel.com/support/performancetools/fortran/linux
Fortran 90	GNU Fortran Compiler v 4.9 or superior	GNU General Public License, GNU Lesser General Public License	http://gcc.gnu.org/fortran/
C/C++	GNU C/C++ compiler v4.9 or superior	GNU General Public License, GNU Lesser General Public License	http://gcc.gnu.org
Java	Oracle Java(TM) Runtime Environment, Standard Edition 1.7 or superior	Oracle Binary Code License Agreement for the Java SE Platform Product	http://www.oracle.com/technetwork/java/index.html

Nevertheless, developers can use their favourite compilers in each case, but this section only provides instructions for using the OSFI libraries with the suggested compilers.

The binary distributions provided assume that the following list of libraries is installed in your system before being able to use OSFI as a shared library. It is not currently possible to build executables that are linked *statically* against the OSFI libraries provided in the binary distribution.

Note that OSFI Source and binary packages include Xerces v3.0 libraries for C/C++.

Table 5: Linux pre-requisites

Component	Purpose	License	Distribution Site
De-compressor	Extract files from release packaged in a compressed tarball	N/A	N/A
Common and C++ version			
libstdc++6 from GCC 4.9 or higher	This package contains an additional runtime library for C++ programs built with the GNU compiler.	LGPL	Linux repository or http://www.gnu.org
libc6 from GCC 4.9 or higher	Contains the standard libraries that are used by nearly all programs on the system. This package includes shared versions of the standard C library and the standard math library, as well as many others.	LGPL	Linux repository or http://www.gnu.org
libgcc1 from GCC 4.9 or higher	Shared version of the support library, a library of internal subroutines that GCC uses to overcome shortcomings of particular machines, or special needs for some languages.	LGPL	Linux repository or http://www.gnu.org

3.3.2.1. IDL

To execute modules in IDL with openSF it is necessary to have installed IDL software on the computer. openSF has been tested with the following versions of this software: version 7.1, 8.0 and 8.1. If the user has a previous version, the application may eventually not work. It is recommended to have installed at least IDL 7.1, and whenever possible version 8.0 or later.

An important requirement for the correct functioning is that IDL is installed in the default path, because if not some features of the OSFI library will not work properly. This problem is related with ConFM module, which uses some internal classes of IDL that must be in the default path, because otherwise the application does not find them. This is caused because IDL looks for these classes only in the default directory, and if it does not find them generates an error.

For IDL 7.1 the default path is '/usr/local/itt/idl' and for IDL 8.x the default path is '/usr/local/itt/idl/idl'.

Furthermore, IDL provides three types of licenses according to the user needs, as can be seen below:

- IDL development: Full license for IDL that allows to the user to use all its functionalities. Users can access to the IDL Development Environment, the IDL command line, and having the ability of compiling and executing IDL .pro files and executing .sav files.
- IDL runtime: Allows executing IDL programs precompiled and saved as .SAV files, or .pro files without any type of restriction.
- IDL virtual machine: It is a free license that allows to the user to execute IDL programs precompiled and saved as .SAV files, or .pro files. This kind of license has a few restrictions, like displaying a splash screen on start-up, callable IDL applications are not available.

To execute a .sav or a .pro file without any type of restriction it is necessary to have installed the development license or the runtime license on the computer. If user wants to generate .sav files by compiling .pro files, it is mandatory to have the development license. If the user only has the virtual machine license, he can execute .pro and .sav files but with restrictions, as many functionalities are not available for this type of license.

3.3.2.2. Matlab

To execute modules in Matlab with openSF Matlab software must be installed on the computer.

The only requirement is that Matlab version must be R2009a or later, with the corresponding license.

3.3.2.3. Python

There are two prerequisites to execute a Python module from openSF:

1. Python correctly installed
2. Environment variable PYTHONPATH configured to point to the necessary libraries (default configuration points to OSFI Python libraries PYTHONPATH=$OSFI_HOME/include/Python).

Python interpreter could be found in the public repositories for the most popular Linux distributions, in Yum system for SUSE/RedHat or Synaptic in Debian/Ubuntu. For further details about installation please visit the Python Project webpage (https://www.python.org)

The OSFI Python libraries are developed to be compliant with both Python 2.X and Python 3.X interpreter. The recommended version is Python 2.7.

3.4. Installation

OSFI comes in different distributions depending on the needs of the user:

- Source package, including necessary sources in every language supported, for including and compiling with other sources.
- Binary package, including headers and static/dynamic libraries for linking with other sources. There is a version of this package for each supported target machine and Operating System.

Figure 1 shows a high-level view of the contents of different OSFI distributions.

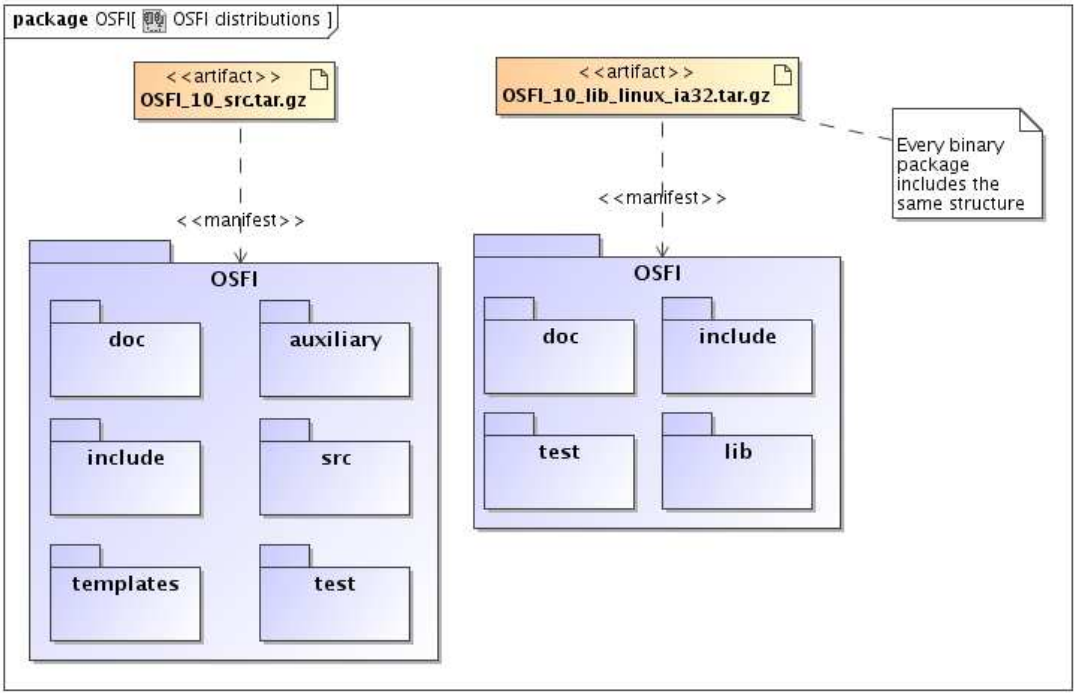

Figure 1: OSFI distributions

To install the OSFI libraries, you shall follow the instructions corresponding to your operating system.

3.4.1. Installation

3.4.1.1. Source Package

First, extract the integration libraries into the desired location and enter it:

```
$ tar –xvzf OSFI-<version>-src.tar.gz
$ cd OSFI
```

Next, run the host OS configuration:

$ make configure-linux	For Linux
$ make configure-osx	For Mac OS X

Now, for the supported system architecture the user must run

$ make configure-ia64	For Intel 64/AMD64 processors

After this action, the C++ Language only users can skip the configuration for Fortran 90 compiler and build libraries and tests running.

```
$ make all-cpp
```

Fortran 90 compiler users must run at this point the configuration rule for their preferred Fortran compiler.

$ make configure-gfortran	For GNU Fortran 90 Compiler developers
$ make configure-ifort	For Intel Fortran Compiler developers

Additional notes for other Fortran compiler users can be found at the end of this section.

Before continue with the installation make sure that the environment variable PATH contains a reference to the folder where Fortran compiler binary files are located. Environment variable PATH can be checked running:

```
$ echo $PATH
```

Users can set this variable in bash shell with:

```
$ export PATH=$PATH:<Full path of Fortran compiler>
```

Optionally users are able to define the path to Fortran compiler with no definition of an environment variable specifying the full path in file $OSFI_HOME/templates/makeDefs. Users should edit FC and FLD fields, which are the pointers to Fortran Compiler and to Fortran Linker respectively. Normally the binary for Fortran Compiler and Linker is the same so FC and FLD should point to the same file.

If users want to develop modules in IDL and execute them in openSF, it is necessary to run the corresponding rule of the makefile.

Furthermore, it is mandatory to configure the environment variable 'IDL_HOME', which must have the full path where IDL is installed on the computer. This environment variable is defined in file $OSFI_HOME/templates/makeDefs, and is configured with the following commands in function of the IDL version. The corresponding command must be executed before the sentence 'make configure-ia32' or 'make configure-ia64'.

```
$ make configure-idl7
```

It configures the environment variable IDL_HOME with the default path for the version 7.x, and sets the necessary libraries to link with this version.

```
$ make configure-idl8
```

It configures the environment variable IDL_HOME with the default path for the version 8.x or later, and sets the necessary libraries to link with this version.

```
$ make idl
```

It generates the executable file for running .pro files.

At this point the user is able to compile the libraries and build the test binaries simply running.

```
$ make all
```

If the installation has been successful, the folder structure should be as follows:

- $OSFI_HOME/src: Source files of the OSFI Libraries
- $OSFI_HOME/lib: Dynamic and static libraries of OSFI
- $OSFI_HOME/include: Header files
- $OSFI_HOME/tests: Source code of the test binaries
- $OSFI_HOME/bin: Test executable files and additional files for testing the correct installation of OSFI Libraries.
- $OSFI_HOME/doc: Documentation of the libraries API in rtf and html format.
- $OSFI_HOME/templates: Definition files for compilation.
- $OSFI_HOME/auxiliary: Folder where Xerces C++ libraries are located

Additional notes for Fortran compiler users

In file $OSFI_HOME/templates/makeDefs are located the definitions used during the compilation and building stage of the OSFI Libraries. Edit it with your preferred text edition tool. Inside the file users will find fields that can be changed.

Relevant fields description for Fortran compiler users:

- FC = Fortran 90 compiler (Example: ifort)
- FLD = Linker used in the Fortran 90 binary building stage (usually this field matches with the Fortran 90 compiler of the FC field)

- FFLAGS = Fortran compiler options. Users of Intel and GNU Fortran compiler are able to add new options as they desire.
- FINC_OPTION = Fortran compiler option which specify the location of the Fortran module files. (Example: Intel Fortran FINC_OPTION= -module; GNU Fortran FINC_OPTION= -J)

3.4.1.2. Binary Package

The binaries package is a set of the libraries and tests, previously build for a particular operating system and architecture. The binaries package must exactly match your current system configuration.

The binary package distributed should be named

```
OSFI-<version>-lib-<operating_system>-<system_arch>.tar.gz
```

Where:

- <version> - the OSFI current version,
- <operating_system> - the target OS, typically linux or osx,
- <system_arch> - the target system architecture and
- <fortran_compiler> - the Fortran 90 compiler used to build the package.

Example:

For Linux Intel Architecture the user must install the OSFI-3.2-lib-linux-ia64.tar.gz package.

For installation extract the files as follows:

```
$ tar –xvzf <OSFI_package_name>.tar.gz
```

The folder structure resultant of this action is as follows

- $OSFI_HOME/lib: Dynamic and static libraries of OSFI
- $OSFI_HOME/include: Header files
- $OSFI_HOME/bin: Test executable files and additional files for testing the correct installation of OSFI Libraries.
- $OSFI_HOME/doc: Documentation of the libraries API in rtf and html format.

4. OPENSF INTEGRATION LIBRARIES

In this section, the following is given:

- An architectural overview, giving structural descriptions of the elements offered in the APIs (such as inheritance diagrams for C++ classes, etc).
- A complete set of examples of how to use the APIs and how to compile and execute them.

4.1. Architectural Overview

The Integration Libraries will serve as interface between the openSF component and the external module, as shown in Figure 2.

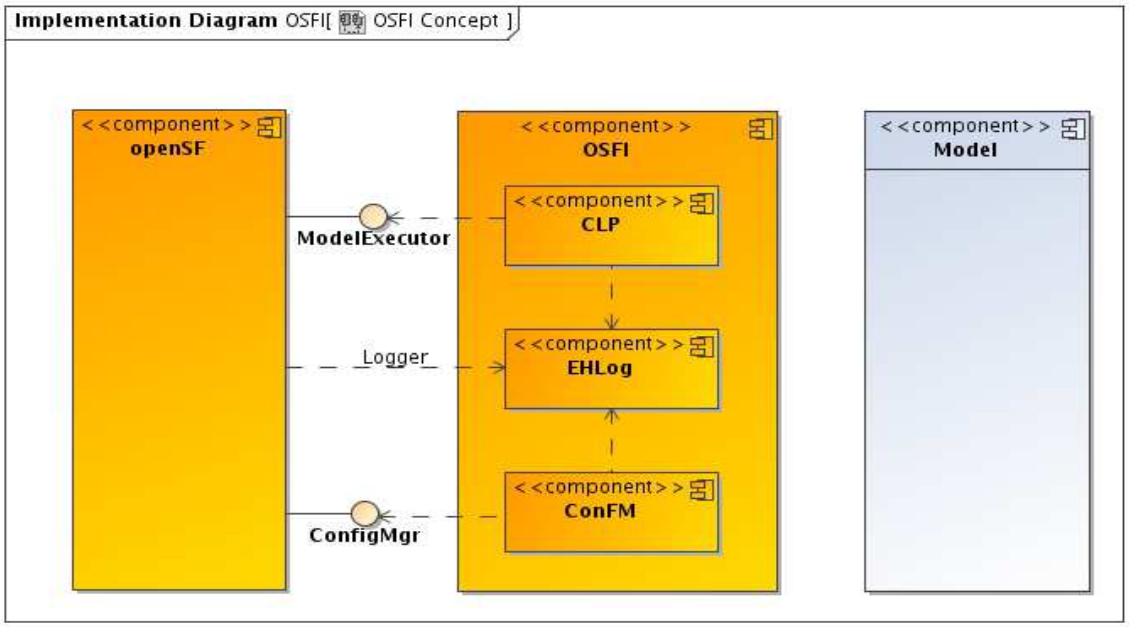

Figure 2: Relationship with openSF and modules

The package "Model" depends on the functionalities implemented in the "OSFI" package.

This package, in turn, depends on the functionalities provided by the "ConfigMgr" and "Model Executor" interfaces of the "openSF" package. This "Model Executor" is the responsible to provide the proper command line arguments for the module execution. The "ConfigMgr" is the module generating the XML configuration files.

There exists a tight integration between the "openSF" package and the "Integration Libraries" package because the former also needs the latter for reading the events raised and logged out from the module execution.

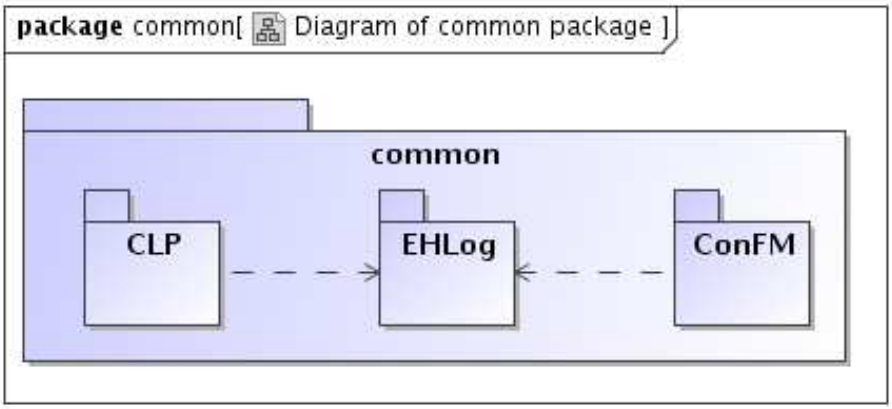

Figure 3 : OSFI common packages

This diagram shows how a system is split up into logical groupings by showing the dependencies among these groupings. As a package is typically thought of as a directory, package diagrams provide a logical hierarchical decomposition of a system.

It can be seen that both "CLP" and "ConFM" packages depend on the "EHLog" package because they are also able to raise certain events to be logged out.

For a deeper analysis of OSFI libraries there is a section in [OSF-ADD] describing the architecture and the interaction with the different programming languages.

4.2. Process logic

In this section, the process logic of using the libraries in modules source code is shown. It is described for C++, Fortran, IDL, Matlab, Python and Java module developers. Note that additional documentation on the APIs available for several languages is available through [OSF-DOC].

4.2.1. C ++ Programming Language

4.2.1.1. CLP

Steps for using the Command Line Parser module:

3. Include the OSFI.h header file in your code
4. Create an instance of the CLP class passing the command line arguments of the main function. The constructor throws an exception in case of error, so remind to catch it.
5. Access the fields with one of the following methods:
   - `getConfFile()`
   - `getConfFiles()`
   - `getInputfiles()`
   - `getOutputFiles()`
6. Destroy the instance once not needed.

4.2.1.2. EHLog

Steps for using the Error Handler and Logging module:

1. Include the OSFI.h header file in your code
2. Use the static functions as a library of functionalities. No instance creation is needed.

4.2.1.3. ConFM

Steps for using the Configuration File Manager module:

1. Include the OSFI.h header file in your code
2. Create an instance of the ParamReader class passing the name of the XML configuration file and, optionally, the name of a XSD schema file to validate it. The constructor throws an exception in case of error, so remind to catch it. Alternatively, you can use the copy constructor to clone an existing instance. Two examples:

```
ParamReader * reader = new ParamReader("config.xml", "");
ParamReader * cloned = new ParamReader(reader);
```

3. Optionally, check for presence of a parameter with a given name. For example:

```
reader->existParameter ("group1.parameter1");
```

4. Access the parameter values by their complete name, using several methods:
   - Directly, with no intermediate instances. For example:

```
reader->getParameter("group1.parameter1").getBooleanValue()
```

   - Accessing the parameters map. For example:

```
t_params_map params = reader->getParameters();
params["group1.parameter1"].print();
```

   - Using an intermediate instance of the Parameter class. For example:

```
Parameter param = reader->getParameter("group1.parameter1");
param.getBooleanValue()
```

   - Accessing the list of parameters under a certain group. For example:

```
vector<Parameter> paramList = reader->getParameters("group1");
for (vector<Parameter>::iterator it = paramList.begin(); it != paramList.end(); ++it)
{
    (*it).print();
}
```

5. Destroy the instance once not needed.

4.2.2. ANSI C Programming Language

The OpenSF integration libraries for C language are designed as a bridge to provide integration between C++ code and C code. Module developers should take account on it because the use of the libraries, most of the times, is as simple as calling a function in module source code.

4.2.2.1. CLP

Steps for using the Command Line Parser module:

1. Include OSFIC.h header file in your code

```
#include "OSFIC.h"
```

2. Initialize the CLP module passing the command line arguments of the main function. The function returns an integer, 1 if the initialisation has been correct 0 otherwise, so remind to catch it.

```
int result = osfiCLP(argc, argv);
```

3. Access the fields with one of the following methods:

```
void osfiCLPGetConfFile(char * fileName, int *length);
void osfiCLPGetConfFiles(char *confFiles[], int *noFiles);
void osfiCLPGetInputFiles(char *inputFiles[], int *noFiles);
void osfiCLPGetOutputFiles(char *outputFiles[], int *noFiles);
```

4. Developers must previously allocate memory for storing file names and file name length.

```
char * fileName = malloc(sizeof(char)*MAX_FILE_ LENGTH NAME);
int length;
osfiCLPGetConfFile(fileName, &length);
```

4.2.2.2. EHLog

Steps for using the Error Handler and Logging module:

1. Include the OSFIC.h header file in your code
2. Use the static functions as a library of functionalities. No instance creation is needed.
3. Methods available in EHLog module are:

```
❑ void osfiLoggerError (char * message);
❑ void osfiLoggerWarning(char *message);
❑ void osfiLoggerInfo(char *message);
❑ void osfiLoggerDebug(char *message);
❑ void osfiLoggerProgress(int n, int m);
❑ void osfiLoggerFinishExecution(int errorCode);
```

4.2.2.3. ConFM

Steps for using the Configuration File Manager module:

1. Include the OSFIC.h header file in your code
2. Create an instance of the ParamReader class passing the name of the XML configuration file and, optionally, the name of a XSD schema file to validate it. The constructor throws an exception in case of error, so remind to catch it. The wrapper for ANSI C returns an integer so developers shall handle this exception through it.

```
int osfiConFMParamReader(char *fileName, char *schemaName);
```

3. Optionally, check for presence of a parameter with a given name.

```
void osfiConFMExistParameter (int *value, char *paramName);
```

4. Access the parameter values by their complete name, using two different methods:
   - Directly with the complete name of the parameter

```
void osfiConFMGetIntegerValue(int *value, char *paramName);
void osfiConFMGetDoubleValue(double *value, char *paramName);
bool osfiConFMGetBoolValue(char *paramName);
void osfiConFMGetFileValue(char *param, int *length, char *paramName);
void osfiConFMGetStringValue(char *param, int *length, char *paramName);
void osfiConFMGetVectorDoubleValues(double *doubleList, int *size,
char *paramName);
void osfiConFMGetVectorIntegerValues(int *intList, int *size, char *paramName);
void osfiConFMGetMatrixDoubleValues(double *doubleMatrix, int *rows,
int *columns, char *paramName);
void osfiConFMGetMatrixIntegerValues(int *intMatrix, int *rows, int *columns,
char *paramName);
```

   - Through a pointer to a osfiParameter structure:

```
typedef struct {
    char *name;
    char *description;
    char *value;
    char *units;
    int dims[2];
    char *min;
    char *max;
} osfiParameter;
void osfiConFMGetParameter(osfiParameter *param, char *paramName);
```

5. Additionally developers can:
   - Prints a textual representation of a parameter or the list of parameters

```
void osfiConFMPrintParameters();
void osfiConFMPrintParameter(char *paramName);
```

- Access to the dimensions of a parameter in order to allocate the accurate memory for parameter values container

```
void osfiConFMGetDimension(char *paramName, int index, int *size);
```

4.2.3. Fortran 90 Programming Language

The OpenSF integration libraries for Fortran language are designed as a bridge to provide integration between C++ code and Fortran code. Module developers should take account on it because the use of the libraries, most of the times, is as simple as calling a function in module source code.

4.2.3.1. CLP

Steps for using the Command Line Parser module

1. Include OSFI.mod Fortran module file in your code

```
Use OSFI
```

2. Check and parse the command line with function osfi_command_line_parser(). This function returns a Logical to check if an exception has been arisen. Remember to handle this exception.

```
Logical :: good
good = osfi_command_line_parser()
if (good) then
    ! Desired actions ...
else
    ! Finish the execution with error code
    call osfi_finish_execution(1)
end if
```

3. Access the fields with the following subroutines
   - `osfi_get_conf_file(confFile)`
   - `osfi_get_input_files(inputFiles)`
   - `osfi_get_output_files(outputFiles)`

```
Character(Len=255)              :: confFile
call osfi_get_conf_file(confFile)
```

4.2.3.2. EHLog

Steps for using the Emergency Handler Log module.

1. Include OSFI.mod Fortran module file in your code
2. Use the static functions for logging a message.

4.2.3.3. ConFM

Steps for using the Configuration File Manager module.

1. Include OSFI.mod Fortran module file in your code

2. Parse the data of the XML configuration file with the function `osfi_param_reader(fileName, schemaName)`. Optionally a XSD schema file can be passed as argument if a validation of the XML file is required. This function returns a Logical to check if an error has happened. Remember to handle those errors.

```
Logical :: good
good = osfi_param_reader('config.xml','')
if ( good) then
      ! Desired actions ...
else
       ! Finish the execution with error code
       call osfi_finish_execution(1)
end if
```

3. Optionally, check for presence of a parameter with a given name.

```
Logical :: exist
exist = osfi_exist_param ('group1.group2.param1')
```

4. Access the parameter values by their complete name, using several subroutines:

```
call osfi_print_parameters()
call osfi_print_parameter(paramName)
```

- Accessing the list of parameters with double matrix structure. For example:

```
Integer :: rows
Integer :: columns
Double precision,dimension(:,:),allocatable :: doubleMatrix
Character(len=50) matrixDoubleParam/"group1.group2.param1"/
call osfi_get_matrix_double_values(doubleMatrix, rows, columns, matrixDoubleParam)
```

4.2.4. Fortran 77 Programming Language

4.2.4.1. CLP

Steps for using the Command Line Parser module.

1. Init the Command Line Parser using the subroutine OCLP()
2. Access the fields with one of the following methods:
   - `OCLPNC(nconf)`: get the number of configuration files
   - `OCLPNI(nin)`: get the number of input files
   - `OCLPNO(nout)`: get the number of output files
   - `OCLPGC(i, fname)`: get configuration file "i"
   - `OCLPGI(i, fname)`: get input file "i"
   - `OCLPGO(i, fname)`: get output file "i"

4.2.4.2. EHLog

Steps for using the Error Handler and Logging module.

1. Use the provided subroutines to generate logs:
   - `OLERR(mess)`: error message
   - `OLINFO(mess)`: information message
   - `OLWAR(mess)`: warning message
   - `OLDEB(mess)`: debug message
   - `OLPROG(n,m)`: progress message (step n of m)
   - `OLFE(errcod)`: finish execution with error code "errcod"
   - `OLQC(vname,value)`: quality with message
   - `OLQD(vname,value)`: quality with double value

4.2.4.3. ConFM

Steps for using the Configuration File Manager module.

1. Initialise the param-reader using the following subroutine:
   - `OPREAD(cnfile,scfile,stat)`
2. Use one of the following subroutines to access the parameter values or properties:
   - `OPEX(pname, pexist)`: check if a parameter exists
   - `OPGPR(pname, rows)`: get number of rows

- `OPGPC(pname, cols)`: get number of columns
- `OPDOUB(dvalue, pname)`: get double parameter
- `OPINT(ivalue, pname)`: get integer parameter
- `OPBOOL(bvalue, pname)`: get boolean parameter
- `OPFILE(fvalue, length, pname)`: get file parameter
- `OPSTR(svalue, length, pname)`: get string parameter
- `OPINV(vector, vsize, pname)`: get integer vector
- `OPDBV(vector, vsize, pname)`: get double vector
- `OPBLV(vector, vsize, pname)`: get boolean vector
- `OPSTRV(vector, vsize, pname)`: get string vector
- `OPFLV(vector, vsize, pname)`: get file vector
- `OPINM(vector, rows, cols, pname)`: get integer matrix
- `OPDBM(vector, rows, cols, pname)`: get double matrix
- `OPBLM(vector, rows, cols, pname)`: get boolean matrix

3. Close param reader:
   - `OPCLS()`

4.2.5. IDL Programming Language

Before using the IDL library for OSFI, it is necessary to compile the corresponding modules: 'CLP.pro', 'Logger.pro', 'Parameter.pro' and 'ConFM.pro' so that all functions are available for IDL.

These files are located in: $OSFI_HOME/include/IDL/

A possible example is:

```
.COMPILE '/home/abma/OSFI/include/IDL/CLP.pro'
.COMPILE '/home/abma/OSFI/include/IDL/Logger.pro'
.COMPILE '/home/abma/OSFI/include/IDL/Parameter.pro'
.COMPILE '/home/abma/OSFI/include/IDL/ConFM.pro'
```

Once these files have been compiled, the developer can define objects of these classes in his own module, and run it.

4.2.5.1. CLP

Steps for using the Command Line Parser module:

1. Create an object of the CLP class passing it as arguments the configuration files, the input files and the output files. It is important to pass these arguments in the correct order.
2. Access the fields with one of the following methods:
   - `getConfFiles()`: Return all the configuration files inside a matrix
   - `getInputFiles()`: Return all the input files inside a matrix
   - `getOutputFiles()`: Return all the output files inside a matrix
   - `getConfFile(index)`: Return the configuration file at the position 'index'.
   - `getInputFile(index)`: Return the input file at the position 'index'.
   - `getOutputFile(index)`: Return the output file at the position 'index'.
3. Destroy the object once not needed.

An example of this procedure is shown below:

```
CLP = OBJ_NEW('CLP', ConfFiles, InputFiles, OutputFiles)
InputFiles = CLP->getInputFiles()
OutputFiles = CLP->getOutputFiles()
ConfFiles = CLP->getConfFiles()
Input = CLP->getInputFile(2)
Output = CLP->getOutputFile(3)
Conf = CLP->getConfFile(0)
OBJ_DESTROY, CLP
```

4.2.5.2. Logger

Steps for using the Logging module:

1. Create an object of the Logger class passing it as argument the debug mode (On=1 or Off=0).
2. Use one of its methods to show different types of messages in stdout:
   - error, message: Shows an error message in openSF format
   - warning, message: Shows a warning message in openSF format
   - info, message: Shows an information message in openSF format
   - debug, message: Shows a debug message in openSF format if debug mode is activated
   - progress, step, nsteps: Shows the progress of the module in openSF format
   - finishExecution: Shows that the module has finished with an information message
   - qualityReport, name, value: Shows a variable and its value
   - setDebugMode, debugMode: Set the debug mode property (On=1, Off=0).
3. Destroy the object once not needed.

An example of this procedure is shown below:

```
LOG = OBJ_NEW('Logger', DebugMode)
LOG->Info, "This is an info message"
LOG->warning, "This is a warning message"
LOG->error, "This is an error message"
LOG->debug, "This is a debug message"
LOG->progress, 2, 21
LOG->qualityReport, 'a', 23
OBJ_DESTROY, LOG
```

4.2.5.3. ConFM

Steps for using the Configuration File Manager module:

1. Create an object of the ConFM class passing the name of the XML configuration file.
2. Optionally, check for a parameter existence with a given name:

   xmlObj->ExistParameter('los.LOS.name')
3. Obtain a parameter of the configuration file by their complete name, using the associated method of ConFM class:

   parameter = GetParameter, path

   This method returns an instance of an object of the Parameter Class.
4. Access the parameter values using several methods:
   - `getPath()`: Returns the path of the parameter
   - `getName()`:Returns the name of the parameter
   - `getDescription()`:Returns the description of the parameter

- `getType()`:Returns the type of the parameter
- `getUnits()`:Returns the units of the parameter
- `getNDims()`:Returns the number of dimensions of the parameter
- `getDims()`:Returns the dimensions of the parameter
- `getValue()`:Returns the value of the parameter
- `getMin()`:Returns the minimum value of the parameter
- `getMax()`:Returns the maximum value of the parameter
- `print`: Shows all the attributes of the parameter in stdout

5. Destroy the objects of classes ConFM and Parameter once not needed.

An example of this procedure is shown below:

```
xmlObj = OBJ_NEW('ConFM', Conf)
xmlPar = xmlObj->GetParameter('los.LOS.name')
print, xmlPar->GetPath()
print, xmlPar->GetValue()
xml->print
OBJ_DESTROY, xmlPar
OBJ_DESTROY, xmlObj
```

4.2.6. Matlab Programming Language

In order to use OSFI in Matlab it is necessary to add the directory containing the OSFI-Matlab files to the search path. It can be easily done by reading the environment variable $OSFI_HOME:

```
% Add OSFI path
OSFI_HOME = getenv('OSFI_HOME');
OSFI_MATLAB = [OSFI_HOME '/include/Matlab/'];
addpath (OSFI_MATLAB);
```

To run the Matlab code from OpenSF, the designed module must be a function with the following structure:

```
function matlabModel (configurationParameters, inputs, outputs)
...
end
```

The following template of a Matlab module is provided to ease the development:

```
function matlabModel (configurationParameters, inputs, outputs)

% Check input arguments
if (nargin<3)
   error ('number of argumets not valid');
end

%---------------------------------------------------------------------
% OSFI Initialization and parameter reading
%---------------------------------------------------------------------
% Add OSFI path
OSFI_HOME = getenv('OSFI_HOME');
OSFI_MATLAB = [OSFI_HOME '/include/Matlab/'];
addpath (OSFI_MATLAB);

% Init CLP and Logger
clp = CLP (configurationParameters, inputs, outputs);
log = Logger ();

log.setDebugMode (true);

% TO-DO: get inputs, outputs and configuration files using
%       CLP getters:
%         + getConfFile(i)
%         + getInputFile(i)
%         + getOutputFile(i)
conf1 = clp.getConfFile (1);

% TO-DO: parse configuration files and read all the parameters
cfm = ConFM (conf1);
%---------------------------------------------------------------------
% Module Processing Core
```

```
%------------------------------------------------------------------
% TO-DO: add code here to perform the processing
end
```

4.2.6.1. CLP

Steps for using the Command Line Parser module:

1. Create an instance of the CLP class passing the configuration parameters, inputs and outputs to the constructor:

```
clp = CLP (configurationParameters, inputs, outputs);
```

2. Access the fields with one of the following methods:
   - `getConfFiles()`
   - `getConfFile(n)`
   - `getInputfiles()`
   - `getInputfile(n)`
   - `getOutputFiles()`
   - `getOutputFile(n)`
   - `nConfFiles()`
   - `nInputFiles()`
   - `nOutputFiles()`

4.2.6.2. EHLog

Steps for using the Error Handler and Logging module:

1. Create an instance of the Logger class

```
log = Logger ();
```

2. Set the debug mode (true or false):

```
log.setDebugMode (true);
```

   If debugMode is false debug messages are not shown.

3. Use the following methods to report logs:
   - `error(message)`
   - `warning(message)`
   - `info(message)`
   - `debug(message)`

- `progress(step, totalSteps)`
- `finishExecution()`
- `qualityReport (name, value)`

```
log.info ('Writing output data');
```

4.2.6.3. ConFM

Steps for using the Configuration File Manager module:

1. Create an instance of the ConFM class passing the name of the configuration file:

```
cfm = ConFM (confFile);
```

2. Optionally, check for presence of a parameter with a given name:

```
exist  = cfm.existParameter ('thresholds.brightness');
```

3. Access the parameter values by their complete name:

```
paramBT  = cfm.getParameter ('thresholds.brightness');
```

4. Each parameter returned by the previous function is an object with several methods that allow accessing to the value and other properties. Use the following methods to read value and properties:
   - `getValue()`
   - `getPath()`
   - `getName()`
   - `getDescription()`
   - `getType()`
   - `getNdims()`
   - `getDims()`
   - `getValue()`
   - `getMin()`
   - `getMax()`

```
BT = paramBT.getValue();
minBT = paramBT.getMin();
maxBT = paramBT.getMax();
...
```

4.2.7. Python Programming Language

In order to use OSFI in Python the Environment variable PYTHONPATH needs to be configured to point to the necessary libraries (default configuration points to OSFI Python libraries PYTHONPATH=$OSFI_HOME/include/Python). This can be done thru the shell or inside a Python module by reading the environment variable $OSFI_HOME:

```
# Set PYTHONPATH
OSFI_HOME = os.environ['OSFI_HOME']
PYTHON_PATH = OSFI_HOME+"/include/Python"
os.environ['PYTHONPATH'] = PYTHON_PATH
```

To run the Python code from OpenSF, the designed module must have the following structure:

```
#!/usr/bin/python

from ParamReader import ParamReader
from CLP import CLP
import Logger

def main(argv=None):
  '''Command line options.'''
...

if __name__ == '__main__':
  main()
```

The following template of a Python module is provided to ease the development:

```
#!/usr/bin/python

from ParamReader import ParamReader
from CLP import CLP
import Logger

def main(argv=None):

  try:

    #--------------------------------------------------------------------
    # OSFI Initialization and parameter reading
    #--------------------------------------------------------------------
    # Set PYTHONPATH
    OSFI_HOME = os.environ['OSFI_HOME']
    PYTHON_PATH = OSFI_HOME+"/include/Matlab"
    os.environ['PYTHONPATH'] = PYTHON_PATH

    # Init CLP
    clp = CLP(argv)
```

```
    # TO-DO: get inputs, outputs and configuration files using
    #       CLP getters:
    #         + getConfFiles()[i]
    #         + getInputFiles()[i]
    #         + getOutputFiles()[i]
    cf = clp.getConfFiles()

    # TO-DO: parse configuration files and read all the parameters
    reader = ParamReader(cf[1], "")

    #--------------------------------------------------------------------
    # Module Processing Core
    #--------------------------------------------------------------------

    # TO-DO: add code here to perform the processing


    #  Finish and process errors
    Logger.finishExecution(0)
    return 0
  except Exception e:
    Logger.error("TestModule failed")
    Logger.finishExecution(1)
    raise(e)

if __name__ == '__main__':
  main()
```

4.2.7.1. CLP

Steps for using the Command Line Parser module:

1. Import the CLP module in your code
2. Create an instance of the CLP class passing the command line arguments of the main function. The constructor throws an exception in case of error, so remind to catch it.
3. Access the fields with one of the following methods:
   - `getConfFile()`
   - `getConfFiles()`
   - `getInputfiles()`
   - `getOutputFiles()`

4.2.7.2. EHLog

Steps for using the Error Handler and Logging module:

1. Import the Logger module in your code
2. Use the static functions as a library of functionalities. No instance creation is needed.

4.2.7.3. ConFM

Steps for using the Configuration File Manager module:

1. Import the ParamReader Python module in your code
2. Create an instance of the ParamReader class passing the name of the XML configuration file and, optionally, the name of a XSD schema file to validate it. The constructor throws an exception in case of error, so remind to catch it:

```
reader = ParamReader("config.xml", "")
```

3. Optionally, check for presence of a parameter with a given name:

```
reader.existParameter("group1.parameter1")
```

4. Access the parameter values by their complete name, using several methods:
   - Directly, with no intermediate instances. For example:

```
reader.getParameter("group1.parameter1").getBooleanValue()
```

   - Accessing the parameters map. For example:

```
params = reader.getAllParameters()
params["group1.parameter1"].write()
```

   - Using an intermediate instance of the Parameter class. For example:

```
param = reader.getParameter("group1.parameter1")
param.getBooleanValue()
```

   - Accessing the list of parameters under a certain group. For example:

```
paramList = reader.getParameters("group1")
for param in paramList:
    param.write()
```

4.2.8. Java Programming Language

4.2.8.1. CLP

Steps for using the Command Line Parser module:

1. Import the osfi.CLP class in your code;
2. Create an instance of the CLP class passing it as parameter the command line arguments of the main function. The constructor throws an exception in case of error, so it should be handled;
3. Access the data fields with one of the following methods:
   - `getConfFile()`
   - `getConfFiles()`
   - `getInputfiles()`
   - `getOutputFiles()`

4.2.8.2. EHLog

Steps for using the Error Handler and Logging module:

1. Import the osfi.Logger class in your code
2. Use the static functions as a library of functionalities. No instance creation is needed.

4.2.8.3. ConFM

Steps for using the Configuration File Manager module:

1. Import the osfi.ParamReader class in your code
2. Create an instance of the ParamReader class passing as argument the name of the XML configuration file and, optionally, the name of a XSD schema file to validate it. The constructor throws an exception in case of error, so it should be handled:

```
ParamReader reader = new ParamReader("config.xml", "");
```

3. Optionally, check for availability of a parameter with a given name:

```
reader.existParameter("group1.parameter1")
```

4. Access the parameter values by their complete name, using several methods:
   - Directly, with no intermediate instances. For example:

```
reader.getParameter("group1.parameter1").getBooleanValue()
```

   - Accessing the parameters map. For example:

```
Map<String, Parameter> params = reader.getAllParameters();
params.get("group1.parameter1").write()
```

- Using an intermediate instance of the Parameter class. For example:

```
Parameter param = reader.getParameter("group1.parameter1")
param.getBooleanValue();
```

- Accessing the list of parameters under a certain group. For example:

```
ArrayList<Parameter> paramList = reader.getParameters("group1");
for (Parameter param : paramList)
    param.write();
```

4.3. Additional Features

4.3.1. Debug Mode

Debug mode logs are activated creating the environment variable "DEBUG_MODE" and setting it to "On". By default if this variable is not present, no debug logs are shown during the execution.

```
export DEBUG_MODE=On
```

4.3.2. Coloured Logs

OSFI provides a mechanism to colour logs when the module is run from command line (only for Unix terminals).

Coloured logs are activated creating the environment variable "OSFI_LOG_COLOR" and setting it to "On".

```
export OSFI_LOG_COLOR=On
```

4.4. Examples of use

4.4.1. C++ Programming Language

Here is an example of C++ code that uses the different modules of the integration libraries.

```
#include "OSFI.h"
#include <iostream>
#include <cstdlib>
#include <list>
#include <string>

using namespace std;

int main(int argc, char * argv[]){

    string matrixIntParam = "los.LOS.polyParY";

    try
    {
        CLP CLP(argc, argv);

        cout << "input files = ";
        for (i = inputFiles.begin(); i != inputFiles.end(); ++i)
            cout << *i << " ";
        cout << endl;

        cout << "output files = ";
        for (i = outputFiles.begin(); i != outputFiles.end(); ++i)
            cout << *i << " ";
        cout << endl;
        string config = CLP.getConfFile();

        // Create a ParamReader
        ParamReader * reader = new ParamReader(config, "");
        Logger::info("Printing whole parameters file");
        reader->print();
        DynamicArray<int> mi;
        mi = reader->getParameter(matrixIntParam).getMatrixInt();
        Logger::info(matrixIntParam);
        for (int i = 0, n = mi.getRows(); i < n; i++)
        {
            for (int j = 0, m = mi.getColumns(); j < m; j++)
            {
                printf("%4d\t", mi[i][j]);
            }
            cout << endl;
        }
```

```
        vector<double> vd = reader-
>getParameter(vectorDoubleParam).getVectorDouble();
        Logger::info(vectorDoubleParam);
        for (int i = 0, n = vd.size(); i < n; i++)
            printf("%4.1f\t", vd[i]);
        cout << endl;
} catch (...)
    {
        Logger::error("Example failed");
        Logger::finishExecution(1);
    }
} // End of main
```

4.4.1.1. C++ Compilation and Execution process.

The compilation process needs to specify the base location of the packages with these environment variables:

- OSFI_HOME, typically < openSF_install_dir>/OSFI

To compile your sources you must specify the location of the header files and the library binaries. This sentence is a valid example for compiling one of the distributed test sources.

```
g++ cppExample.cpp –o cppExample –I$OSFI_HOME/include –L$OSFI_HOME/lib
–losfi-common –lxerces-c
```

Integration libraries come in two distribution types, shared or static libraries.

If you have linked the shared libraries you can execute the binary files after specifying the location of those shared libraries like this:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$OSFI_HOME/lib
```

Linking with static libraries does not imply to specify the location of the linked libraries since the executable already includes the object files.

The sentences for executing the test binaries are:

```
./cppExample <arguments>
```

4.4.2. ANSI C Programming Language

Here is an example of ANSI C code that uses the different modules of the integration libraries.

```
#include "OSFIC.h" // Include OSFI header for ANSI C Programming Language
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

    char* doubleParam = "los.LOS.initialTime.second";
    char* matrixDoubleParam = "sensor.NumericModel.polyParX";

    osfiCLP(argc, argv); // Initialize CLP component

    if (osfiConFMParamReader(configurationFile, "") == 1) {

        osfiLoggerInfo("ConFM::Configuration File Parsed");

        double doubleParameter;
        osfiConFMGetDoubleValue(&doubleParameter, doubleParam);

        // Matrix Parameters
        int x, y;
        osfiConFMGetDimension(matrixDoubleParam, 1, &x);
        osfiConFMGetDimension(matrixDoubleParam, 2, &y);
        double *doubleMatrix = malloc(x * y * sizeof(double));
        int rows;
        int columns;
        osfiConFMGetMatrixDoubleValues(doubleMatrix, &rows, &columns,
                matrixDoubleParam);
        for (i = 0; i < rows; i++) {
            for (j = 0; j < columns; j++) {
                printf("Matrix Double %s [%d[%d]= %f\n", matrixDoubleParam, i,
                        j, doubleMatrix[(i * columns) + j]);
            }
        }
        free(doubleMatrix);

        //Parameter Structure Retrieval
        osfiParameter * param = malloc(sizeof(osfiParameter));
        param->name = malloc(MAX_PARAMETER_NAME_SIZE * sizeof(char));
        param->description = malloc(MAX_PARAMETER_DESC_SIZE * sizeof(char));
        param->value = malloc(MAX_PARAMETER_FIELD_SIZE * sizeof(char));
        param->units = malloc(MAX_PARAMETER_FIELD_SIZE * sizeof(char));
        param->min = malloc(MAX_PARAMETER_FIELD_SIZE * sizeof(char));
        param->max = malloc(MAX_PARAMETER_FIELD_SIZE * sizeof(char));
        osfiConFMGetParameter(param, matrixDoubleParam);
        free(param->name);
        free(param->description);
        free(param->value);
```

```
        free(param->units);
        free(param->min);
        free(param->max);
        free(param);
    } else {
        osfiLoggerError("ConFM::Error Loading Configuration File\n");
    }
    osfiLoggerFinishExecution(0);
    return 0;
}
```

4.4.2.1. ANSI C Compilation and Execution process

The compilation process needs to specify the base location of the packages with these environment variables:

- OSFI_HOME, typically < openSF_install_dir>/OSFI

To compile your sources you must specify the location of the header files and the library binaries. This sentence is a valid example for compiling one of the distributed test sources.

```
gcc cExample.c –o cExample –I$OSFI_HOME/include –L$OSFI_HOME/lib
–losfi-common-c –lxerces-c
```

Integration libraries come in two distribution types, shared or static libraries.

If you have linked the shared libraries you can execute the binary files after specifying the location of those shared libraries like this:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$OSFI_HOME/lib
```

Linking with static libraries does not imply to specify the location of the linked libraries since the executable already includes the object files.

The sentences for executing the test binaries are:

./cExample <arguments>

4.4.3. Fortran 90 Programming Language

Here is an example of Fortran code that uses the different modules of the integration libraries.

```
Program f90Example

! Include the OSFI module
Use OSFI

    Character(Len=255)                    :: confFile
    Character(Len=255), dimension(:), allocatable :: inputFiles
    Character(Len=255), dimension(:), allocatable :: outputFiles
    Character(len=50) matrixDoubleParam/"sensor.NumericModel.polyParX"/
    Character(Len=2)                      :: tmp
    Double precision,dimension(:,:),allocatable :: doubleMatrix
    Integer                               :: arraySize
    Integer                               :: index
    Integer :: i
    Integer :: j
    Integer :: rows
    Integer :: columns
    Logical :: good

    good = osfi_command_line_parser()
    if ( good) then

        ! Parse config file
        call osfi_get_conf_file(confFile)
        call osfi_info("Configuration file = " // adjustl(trim(confFile)))

        ! Parse input files
        call osfi_get_input_files(inputFiles)
        arraySize = size(inputFiles, 1)
        index = 1
        do while (index .LE. arraySize)
            write (tmp, '(I1)') index
            call osfi_info("Input file[" // trim(tmp) // "] = " //
adjustl(trim(inputFiles(index))))
            index = index + 1
        end do

        ! Parse output files
        call osfi_get_output_files(outputFiles)
        arraySize = size(outputFiles, 1)
        index = 1
        do while (index .LE. arraySize)
            write (tmp, '(I1)') index
            call osfi_info("Output file[" // trim(tmp) // "] = " //
adjustl(trim(outputFiles(index))))
            index = index + 1
        end do
```

```
        ! ConFM Module Example
        good = osfi_param_reader(confFile, '')
        if ( good ) then
            call osfi_info("Printing whole parameters file")
            call osfi_print_parameters()
            call osfi_get_matrix_double_values(doubleMatrix, rows, columns,
matrixDoubleParam)
            call osfi_info(matrixDoubleParam)

            i = 1
            do while (i .LE. columns)
                j = 1
                do while (j .LE. rows)
                    write (6, '(F4.1,TR4)', ADVANCE='NO') doubleMatrix(j, i)
                    j = j + 1
                end do
                write (6, *) ""
                i = i + 1
            end do
        else
            ! Finish the execution In case an exception has been arisen
            call osfi_error('f90Example Failed')
            call osfi_finish_execution(1)
        end if

        call osfi_finish_execution(0)
    else
        ! Finish the execution In case an exception has been arisen
        call osfi_error('f90Example Failed')
        call osfi_finish_execution(1)
    end if
end
```

4.4.3.1. Fortran 90 Compilation and Execution instructions.

The compilation process needs to specify the base location of the packages with these environment variables:

- OSFI_HOME, typically <openSF_install_dir>/OSFI

To compile your sources you must specify the location of the header files and the library binaries. This sentence is a valid example for compiling one of the distributed test sources.

In this case Intel Fortran Compiler has been chosen as default Fortran compiler.

```
ifort f90Example.f90 –o f90Example –I$OSFI_HOME/include –L$OSFI_HOME/lib
–losfi-common –losfi-f90
```

Integration libraries come in two distribution types, shared or static libraries.

If you have linked the shared libraries you can execute the binary files after specifying the location of those shared libraries like this:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$OSFI_HOME/lib
```

Linking with static libraries does not imply to specify the location of the linked libraries since the executable already includes the object files.

The sentences for executing the test binaries are:

```
./f90Example <arguments>
```

4.4.4. Fortran 77 Programming Language

Here is an example of Fortran 77 code that uses the different modules of the integration libraries.

```
      program test
      implicit none

      INTEGER*4 n,m,errcod,stat
      CHARACTER*255 cc
      CHARACTER*32 vname
      CHARACTER*32 vvalue
      CHARACTER*32 cnfile
      CHARACTER*32 scfile
      REAL*8 value,dvalue
      CHARACTER*32 pname
      Character*66 tmp
      CHARACTER*255 fname
      INTEGER*1 nconf, nin, nout, i

c---- ConFM param variables
      INTEGER nrows, ncols, length, vsize, k, j, p
      LOGICAL*1 pexist, bvalue
      INTEGER ivalue
      CHARACTER*255 fvalue, svalue
      INTEGER ivector(25)
      REAL*8 dvector(25)
      LOGICAL bvector(25)
      CHARACTER svector(255)*255, fvector(255)*255
      INTEGER imatrix(255)
      REAL*8 dmatrix(255)
      LOGICAL*1 bmatrix(255)

c -------------------------------------------------
c     TEST OSFI COMMAND LINE PARSER
c -------------------------------------------------
      WRITE (*,*) ''
      call OLINFO ('----------------------------')
      call OLINFO ('     TESTING CLP')
      call OLINFO ('----------------------------')

c      call OCLP(stat)

c    Get number of configuration files
      call OCLPNC (nconf)
      WRITE(tmp, '(I2)') nconf
      call OLINFO ('Number of configuration files: '//tmp)

c    Get number of configuration files
      call OCLPNI (nin)
      WRITE (tmp,'(I2)') nin
      call OLINFO ('Number of input files/folders: '//tmp)
```

```
c    Get number of configuration files
     call OCLPNO (nout)
     WRITE (tmp,'(I2)') nout
     call OLINFO ('Number of output files/folders: '//tmp)

c    Print configuration files
     i = 1
     DO WHILE (i.NE.nconf+1)
       call OCLPGC (i, fname)
       WRITE (tmp,'(I2)') i
       call OLINFO ('Configuration file '//tmp(1:4)//': '//fname)

       i = i + 1
     END DO

c    Print input files/folders
     i = 1
     DO WHILE (i.NE.nin+1)
       call OCLPGI (i, fname)
       WRITE (tmp,'(I2)') i
       call OLINFO ('Input file/folder  '//tmp(1:4)//': '//fname)

       i = i + 1
     END DO

c    Print output files/folders
     i = 1
     DO WHILE (i.NE.nout+1)
       call OCLPGO (i, fname)
       WRITE (tmp,'(I2)') i
       call OLINFO ('Output file/folder '//tmp(1:4)//': '//fname)

       i = i + 1
     END DO

c ------------------------------------------------------
c     TEST OSFI LOGGER
c ------------------------------------------------------
     WRITE (*,*) ''
     call OLINFO ('----------------------------')
     call OLINFO ('     TESTING LOGGER')
     call OLINFO ('----------------------------')

     cc = 'Test OSFI Info Message'
     call OLINFO(cc)

     call OLINFO ('This is another Info Message')

     n = 3
     m = 4
     call OLPROG(n,m)

     cc = 'Test OSFI Error Message'
```

```
      call OLERR(cc)

      cc = 'Test OSFI Warning Message'
      call OLWAR(cc)

      vname = 'QualityVariable'
      vvalue = 'Good'
      call OLQC(vname,vvalue)

      value = 10.5
      call OLQD(vname,value)

      cnfile = 'exampleFile.xml'
      scfile = ''

      cc = 'Test OSFI Debug Message'
      call OLDEB(cc)

c -----------------------------------------------------
c     TEST OSFI PARAM READING
c -----------------------------------------------------
      WRITE (*,*) ''
      call OLINFO ('---------------------------')
      call OLINFO ('     TESTING PARAM READING')
      call OLINFO ('---------------------------')

c---- Param reader initialization
      call OPREAD(cnfile,scfile,stat)
      if (stat.NE.1) then
        cc = 'Error Parsing '//trim(cnfile)
        call OLERR(cc)
      end if

      call OLINFO('')
      call OLINFO('Getting cols and rows for los.LOS.polyParY:')
c---- Get param rows
      pname = 'los.LOS.polyParY'//CHAR(0)
      call OPGPR (pname, nrows)
      WRITE(tmp, *) nrows
      call OLINFO('   Rows: '//tmp)

c---- Get param cols
      pname = 'los.LOS.polyParY'//CHAR(0)
      call OPGPC (pname, ncols)
      WRITE(tmp, *) ncols
      call OLINFO('   Cols: '//tmp)



c---- Check if parameter exist
      call OLINFO('')
      call OLINFO('Testing if a parameter exists or not:')

      pname = 'los.LOS.polyParY'//CHAR(0)
```

```
      call OPEX (pname, pexist)
      WRITE (tmp,*) pexist
      call OLINFO('   los.LOS.polyParY:    '//tmp)

      pname = 'this.does.not.exist'//CHAR(0)
      call OPEX (pname, pexist)
      WRITE (tmp,*) pexist
      call OLINFO('   this.does.not.exist: '//tmp)



c---- READING SCALAR PARAMETERS
c-----------------------------------------------------------------

      call OLINFO('')
      call OLINFO('')
c---- Get integer value
      pname = 'earth.Earth.demType'
      call OPINT(ivalue, pname)
      WRITE(tmp, *) ivalue
      cc = 'Integer parameter '//TRIM(pname)//' = '//tmp
      call OLINFO(cc)

c---- Get boolean parameter
      pname = 'sensor.Sensor.isRPY'
      call OPBOOL(bvalue, pname)
      WRITE(tmp, *) bvalue
      cc = 'Boolean parameter '//TRIM(pname)//' = '//tmp
      call OLINFO(cc)

c---- Get double parameter
      pname = 'los.LOS.initialTime.second'
      call OPDOUB(dvalue, pname)
      WRITE(tmp, '(F12.5)') dvalue
      cc = 'Double parameter '//TRIM(pname)//' = '//tmp
      call OLINFO(cc)

c---- Get file parameter
      pname = 'files.singleFile'
      call OPFILE(fvalue, length, pname)
      cc = 'File parameter '//TRIM(pname)//' = '//fvalue
      call OLINFO(cc)

c---- Get string parameter
      pname = 'sensor.name'
      call OPSTR(svalue, length, pname)
      cc = 'String parameter '//TRIM(pname)//' = '//svalue
      call OLINFO(cc)

c---- READING VECTOR PARAMETERS
c-----------------------------------------------------------------

c---- Get integer vector
      pname = 'los.LOS.intVector'
```

```
      call OPINV (ivector, vsize, pname)
      WRITE (tmp,*) ivector (1:vsize)
      cc = 'Integer vector '//TRIM(pname)//' = '//tmp
      call OLINFO(cc)

c---- Get double vector
      pname = 'los.LOS.doubleVector'
      call OPDBV (dvector, vsize, pname)
      cc = 'Double vector '//TRIM(pname)//':'
      call OLINFO(cc)
      WRITE (*, '(F12.1)') dvector (1:vsize)

c---- Get boolean vector
      pname = 'los.LOS.flagsVector'
      call OPBLV (bvector, vsize, pname)
      WRITE (tmp, *) bvector (1:vsize)
      cc = 'Boolean vector '//TRIM(pname)//':'//tmp
      call OLINFO(cc)

c---- Get string vector
      pname = 'los.LOS.missionNames'
      call OPSTRV (svector, vsize, pname)
      call OLINFO ('String vector '//TRIM(pname)//':')
      DO k=1,vsize
        WRITE (*,*) svector(k)
      END DO

c---- Get file vector
      pname = 'los.LOS.orbitFiles'
      call OPFLV (fvector, vsize, pname)
      call OLINFO ('File vector '//TRIM(pname)//':')
      DO k=1,vsize
        WRITE (*,*) fvector(k)
      END DO

c---- READING MATRIX PARAMETERS
c-----------------------------------------------------------------------

c---- Get integer matrix
      pname = 'los.LOS.polyParY'
      call OPINM (imatrix, nrows, ncols, pname)
      cc = 'Integer matrix '//TRIM(pname)//':'
      call OLINFO(cc)
      k = 0
      DO WHILE (k.NE.nrows)
        j = 0
        DO WHILE (j.NE.ncols)
          p = k*ncols + j
          WRITE (*,10) 'row=', k, ' col=', j, ' ==> ', imatrix(p+1)
 10       FORMAT(A4,I1,A5,I1,A5,I3)
          j = j + 1
        END DO
      k = k + 1;
      END DO
```

```
c---- Get double matrix
    pname = 'sensor.NumericModel.polyParX'
    call OPDBM (dmatrix, nrows, ncols, pname)
    cc = 'Double matrix '//TRIM(pname)//':'
    call OLINFO(cc)
    k = 0
    DO WHILE (k.NE.nrows)
      j = 0
      DO WHILE (j.NE.ncols)
        p = k*ncols + j
        WRITE (*,15) 'row=', k, ' col=', j, ' ==> ', dmatrix(p+1)
 15       FORMAT(A4,I1,A5,I1,A5,F8.2)
        j = j + 1
      END DO
    k = k + 1;
    END DO

c---- Get boolean matrix
    pname = 'los.LOS.flagsMatrix'
    call OPBLM (bmatrix, nrows, ncols, pname)
    cc = 'Boolean matrix '//TRIM(pname)//':'
    call OLINFO(cc)
    k = 0
    DO WHILE (k.NE.nrows)
      j = 0
      DO WHILE (j.NE.ncols)
        p = k*ncols + j
        WRITE (*,20) 'row=', k, ' col=', j, ' ==> ', bmatrix(p+1)
 20       FORMAT(A4,I1,A5,I1,A5,L)
        j = j + 1
      END DO
    k = k + 1;
    END DO

c---- Close OSFI param-reader
    call OPCLS()

c---- Exit with error code
    errcod = 0
    call OLFE(errcod)
    end
```

4.4.4.1. Fortran 77 Compilation and Execution Process

The compilation process needs to specify the base location of the packages with these environment variables:

- OSFI_HOME, typically < openSF_install_dir>/OSFI

To compile your sources you must specify the location of the header files and the library binaries. This sentence is a valid example for compiling one of the distributed test sources.

```
gfortran -o test test.f –losfi-f77 -losfi-common-c -losfi-common -lxerces-c –
L$OSFI_HOME/lib
```

Integration libraries come in two distribution types, shared or static libraries.

If you have linked the shared libraries you can execute the binary files after specifying the location of those shared libraries like this:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$OSFI_HOME/lib
```

Linking with static libraries does not imply to specify the location of the linked libraries since the executable already includes the object files.

The sentences for executing the test binaries are:

```
./test <arguments>
```

4.4.5. IDL Programming Language

Here is an example of IDL code that uses the different modules of the integration libraries.

```
; openSF Integration Libraries (OSFI)

PRO test_IDL, ConfFiles, InputFiles, OutputFiles, DebugMode

IF N_PARAMS() LT 3 THEN BEGIN
 EXECUTION_MODE = GETENV('IDL_EXECUTION_MODE')
 IF (STRCMP(EXECUTION_MODE, 'SAV') NE 1) THEN $
  print, 'Number of arguments not valid'
ENDIF

IF N_PARAMS() EQ 3 THEN $
 DebugMode = 0

;Show some logs
print, ''
print, 'Show some logs examples using Logger class...'
LOG = OBJ_NEW('Logger', DebugMode)
LOG->Info, "This is an info message"
LOG->warning, "This is a warning message"
LOG->debug, "This is a debug message"
LOG->progress, 2, 21
LOG->qualityReport, 'a', 23

;Show configuration files, inputs and outputs using CLP
print, ''
print, 'Parsing configuration, input and output files using CLP class...'
CLP = OBJ_NEW('CLP', ConfFiles, InputFiles, OutputFiles)
InputFiles = CLP->GetInputFiles()
OutputFiles = CLP->GetOutputFiles()
ConfFiles = CLP->GetConfFiles()
Input = CLP->GetInputFile(2)
Output = CLP->GetOutputFile(3)
IF (N_ELEMENTS(ConfFiles) EQ 1) THEN BEGIN
 Conf = CLP->getConfFile(0)
ENDIF ELSE BEGIN
 Conf = CLP->getConfFile(1)
ENDELSE

LOG->Info, "Configuration files: " + ConfFiles
LOG->Info, "Input files: " + InputFiles
LOG->Info, "Output files: " + OutputFiles
LOG->Info, "Configuration file: " + Conf
LOG->Info, "Input file: " + Input
LOG->Info, "Output file: " + Output
 success = 1

;Parse XML file and check read values
print, ''
```

```
print, 'Parsing XML file and checking that read values are correct...'
xmlObj = OBJ_NEW('ConFM', Conf)

xmlPar = xmlObj->GetParameter('los.LOS.name')
IF (STRCMP(xmlPar->GetValue(), 'my LOS') EQ 1) THEN BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
ENDIF ELSE BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
 success = 0
ENDELSE

xmlPar = xmlObj->GetParameter('los.LOS.polyParY')
value = xmlPar->GetValue()
result = [1,2,3,4,5,6,7,8,9,10,11,12]
IF max(value-result) EQ min(value-result) THEN BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
ENDIF ELSE BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
 success = 0
ENDELSE

xmlPar = xmlObj->GetParameter('los.LOS.initialTime.year')
value = xmlPar->GetValue()
IF value EQ 2009 THEN BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
ENDIF ELSE BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
 success = 0
ENDELSE

xmlPar = xmlObj->GetParameter('los.LOS.initialTime.second')
value = xmlPar->GetValue()
IF value EQ 1.0 THEN BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
ENDIF ELSE BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
 success = 0
ENDELSE

xmlPar = xmlObj->GetParameter('los.LOS.iDomain')
value = xmlPar->GetValue()
result = [0.0,3.0]
IF max(value-result) EQ min(value-result)  THEN BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
ENDIF ELSE BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
 success = 0
ENDELSE

xmlPar = xmlObj->GetParameter('los.LOS.flagsMatrix')
value = xmlPar->GetValue()
result = ['TRUE','FALSE','TRUE','FALSE']
IF where(strcmp(value, result) NE 1) EQ -1 THEN BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
```

```
ENDIF ELSE BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
 success = 0
ENDELSE

xmlPar = xmlObj->GetParameter('los.LOS.flagsVector')
value = xmlPar->GetValue()
result = ['TRUE','TRUE']
IF where(strcmp(value, result) NE 1) EQ -1 THEN BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
ENDIF ELSE BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
 success = 0
ENDELSE

xmlPar = xmlObj->GetParameter('los.LOS.instrumentTypes')
value = xmlPar->GetValue()
result = ['Dummy','Camera','Lidar','SAR']
IF where(strcmp(value, result) NE 1) EQ -1 THEN BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
ENDIF ELSE BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
 success = 0
ENDELSE

xmlPar = xmlObj->GetParameter('los.LOS.missionNames')
value = xmlPar->GetValue()
result = ['BioMass','Premier','CoreH2O']
IF where(strcmp(value, result) NE 1) EQ -1 THEN BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
ENDIF ELSE BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
 success = 0
ENDELSE

xmlPar = xmlObj->GetParameter('los.LOS.detectorFiles')
value = xmlPar->GetValue()
result = ['/home/det1.xml','det2.xml','C:/home/det3.xml','det4.xml']
IF where(strcmp(value, result) NE 1) EQ -1 THEN BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
ENDIF ELSE BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
 success = 0
ENDELSE

xmlPar = xmlObj->GetParameter('los.LOS.orbitFiles')
value = xmlPar->GetValue()
result = ['data/sunSync.dat','data/geoSync.dat']
IF where(strcmp(value, result) NE 1) EQ -1 THEN BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
ENDIF ELSE BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
 success = 0
ENDELSE
```

```
xmlPar = xmlObj->GetParameter('files.singleFile')
IF (STRCMP(xmlPar->GetValue(), 'exampleFile.xml') EQ 1) THEN BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
ENDIF ELSE BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
 success = 0
ENDELSE

xmlPar = xmlObj->GetParameter('files.vectorFiles')
value = xmlPar->GetValue()
result = ['exampleFile.xml','test.xml']
IF where(strcmp(value, result) NE 1) EQ -1 THEN BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
ENDIF ELSE BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
 success = 0
ENDELSE

xmlPar = xmlObj->GetParameter('files.matrixFiles')
value = xmlPar->GetValue()
result = ['test.xml','test.xml','test.xml','exampleFile.xml']
IF where(strcmp(value, result) NE 1) EQ -1 THEN BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
ENDIF ELSE BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
 success = 0
ENDELSE

xmlPar = xmlObj->GetParameter('sensor.name')
IF (STRCMP(xmlPar->GetValue(), 'my LOS') EQ 1) THEN BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
ENDIF ELSE BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
 success = 0
ENDELSE

xmlPar = xmlObj->GetParameter('sensor.Sensor.isRPY')
IF (STRCMP(xmlPar->GetValue(), 'TRUE') EQ 1) THEN BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
ENDIF ELSE BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
 success = 0
ENDELSE

xmlPar = xmlObj->GetParameter('sensor.Detector.size')
value = xmlPar->GetValue()
result = [5,5]
IF max(value-result) EQ min(value-result)  THEN BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
ENDIF ELSE BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
 success = 0
ENDELSE
```

```
xmlPar = xmlObj->GetParameter('sensor.NumericModel.polyParX')
value = xmlPar->GetValue()
result = [1.1,2.2,3.3,4.4,5.5,6.1,7.2,8.3,9.4]
IF max(value-result) EQ min(value-result)  THEN BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
ENDIF ELSE BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
 success = 0
ENDELSE

xmlPar = xmlObj->GetParameter('sensor.SensitivityMatrix.sensorFile')
IF (STRCMP(xmlPar->GetValue(), 'models/LOSM/conf/sensor.xml') EQ 1) THEN BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
ENDIF ELSE BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
 success = 0
ENDELSE

xmlPar = xmlObj->GetParameter('earth.Earth.demType')
value = xmlPar->GetValue()
IF value EQ 0 THEN BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
ENDIF ELSE BEGIN
 print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
 success = 0
ENDELSE

OBJ_DESTROY, xmlPar
OBJ_DESTROY, xmlObj
OBJ_DESTROY, CLP
OBJ_DESTROY, LOG

print, ''

IF success EQ 1 THEN $
 print, 'Successful test' $
ELSE $
 print, 'Failed test'

END
```

4.4.5.1. IDL licenses

IDL provides three types of licenses in function of the needs of the user:

- IDL development: Full license for IDL that allows to the user to use all its functionalities. Users can access to the IDL Development Environment, the IDL command line, and having the ability of compiling and executing IDL .pro files.

- IDL runtime: Allows executing IDL programs precompiled and saved as .SAV files without any type of restriction.
- IDL virtual machine: It is a free license that allows to the user to execute IDL programs precompiled and saved as .SAV files. This kind of license has a few restrictions, like displaying a splash screen on startup, callable IDL applications are not available...

4.4.6. Matlab programming language

Here is an example of Matlab code that uses the different modules of the integration libraries.

```
function CloudsDetection (configurationParameters, inputs, outputs)

% Check input arguments
if (nargin<3)
   error ('number of argumets not valid');
end

%--------------------------------------------------------------------
% OSFI Initialization and parameter reading
%--------------------------------------------------------------------
% Add OSFI path
OSFI_HOME = getenv('OSFI_HOME');
OSFI_MATLAB = [OSFI_HOME '/include/Matlab/'];
addpath (OSFI_MATLAB);

% Init CLP and Logger
clp = CLP (configurationParameters, inputs, outputs);
log = Logger ();
log.setDebugMode (true);

% Get inputs, outputs and configuration files using
inputFolder = clp.getInputFile (1);
outFile     = clp.getOutputFile (1);
confFile    = clp.getConfFile (1);

% Parse configuration files and read all the parameters
log.info (['Reading configuration parameters from ' confFile]);
cfm = ConFM (confFile);

brightness_threshold  = cfm.getParameter ('thresholds.brightness').getValue;
NDSI_threshold        = cfm.getParameter ('thresholds.NDSI').getValue;
temperature_threshold = cfm.getParameter ('thresholds.temperature').getValue;
composite_threshold   = cfm.getParameter ('thresholds.composite').getValue;
filter5_threshold     = cfm.getParameter ('thresholds.filter5').getValue;
filter6_threshold     = cfm.getParameter ('thresholds.filter6').getValue;
filter7_threshold     = cfm.getParameter ('thresholds.filter7').getValue;
filter8_threshold     = cfm.getParameter ('thresholds.filter8').getValue;

%--------------------------------------------------------------------
% Module Processing Core
%--------------------------------------------------------------------
% Read input images
log.info ('Reading input files');
BLUE  = imread ([inputFolder '/B10.TIF']); % blue-green band
GREEN = imread ([inputFolder '/B20.TIF']); % green
RED   = imread ([inputFolder '/B30.TIF']); % red
NIR   = imread ([inputFolder '/B40.TIF']); % near infrared
MIR1  = imread ([inputFolder '/B50.TIF']); % mid-infrared
```

```
TIR   = imread ([inputFolder '/B60.TIF']); % thermal infrared
MIR2  = imread ([inputFolder '/B70.TIF']); % mid-infrared
[rows cols] = size (BLUE);

% Process images
log.info ('Processing images');
OUT = [];
NDSI     = (GREEN - MIR1)./(GREEN + MIR1);
composite = (1 - MIR1).*TIR;
filter5   = NIR./RED;
filter6   = NIR./GREEN;
filter7   = NIR./MIR1;
filter8   = MIR1./TIR;

NO_CLOUD   =
(RED<brightness_threshold)|(NDSI>NDSI_threshold)|(TIR>temperature_threshold);
AMBIGUOUS  =
((composite>composite_threshold)|(filter5>filter5_threshold)|(filter6>filter6_threshold)|
(filter7<filter7_threshold));
WARM_CLOUD = (filter8>filter8_threshold);
COLD_CLOUD = (filter8<=filter8_threshold);

OUT = AMBIGUOUS*50;
pos = find (OUT==0);
OUT(pos) = WARM_CLOUD(pos)*150 + COLD_CLOUD(pos)*255;
OUT = OUT.*not(NO_CLOUD);

% Write data
log.info ('Writing output data');
imwrite (uint8(OUT), outFile);

end
```

4.4.6.1. Matlab Compilation and Execution Process

Since Matlab is an interpreted language it can't be compiled. There are only two prerequisite to execute a model from OpenSF:

1. Matlab
2. Each model must be a function with the following format (see section 4.2.5):

```
function matlabModel (configurationParameters, inputs, outputs)
...
end
```

4.4.7. Python Programming Language

Here is an example of Python code that uses the different modules of the integration libraries.

```
#!/usr/bin/python

from ParamReader import ParamReader
from CLP import CLP
import Logger

def main(argv=None):
  matrixIntParam = 'los.LOS.polyParY'
  vectorDoubleParam = 'los.LOS.iDomain'

  try:

    clp = CLP(argv)

    # Show conf files, inputs and outputs using CLP
    cf = clp.getConfFiles()
    Logger.info ('Configuration files: ' +  ', '.join(cf))
    inf = clp.getInputFiles ()
    Logger.info ('Input files:       ' + ', '.join(inf))
    outf = clp.getOutputFiles ()
    Logger.info ('Output files:      ' + ', '.join(outf))

    # Local parameters reading
    reader = ParamReader(cf[1], "")
    Logger.info("Printing whole parameters file")
    reader.write()

    mi = reader.getParameter(matrixIntParam).getMatrixInt()
    Logger.info(matrixIntParam)
    for i in range(len(mi)):
      for j in range(len(mi[0])):
        print "[%s][%s] = %s" % (i, j, mi[i][j])

    vd = reader.getParameter(vectorDoubleParam).getVectorDouble()
    Logger.getInfoStream().write(vectorDoubleParam)
    print("\n")
    for d in vd:
      print(d)

    Logger.finishExecution(0)
    return 0
  except Exception:
    Logger.error("TestModule failed")
    Logger.finishExecution(1)
```

4.4.7.1. Python Compilation and Execution process.

Since Python is an interpreted language it can't be compiled. There are only two prerequisite to execute a model from OpenSF:

1. Python correctly installed
2. Environment variable PYTHONPATH configured to point to the necessary libraries (default configuration points to OSFI Python libraries PYTHONPATH=$OSFI_HOME/include/Python)

4.4.8. Java Programming Language

Below is an example of Java code that uses the different modules of the integration libraries.

```
import java.util.ArrayList;

import osfi.CLP;
import osfi.Logger;
import osfi.ParamReader;
import osfi.Parameter;

public class TestModel {

    public static void main(String[] args) {

        try {
            CLP clp = new CLP(args);
            ArrayList<String> cf = clp.getConfFiles();

            Logger.info ("This is an info message");
            Logger.warning ("This is a warning message");
            Logger.debug ("This is a debug message");

            ParamReader cfm = new ParamReader(cf.get(1),"");
            Logger.info ("Configuration files: " +   CLP.arrayToString(cf));
            ArrayList<String> inf = clp.getInputFiles();
            Logger.info ("Input files:       " + CLP.arrayToString(inf));
            ArrayList<String> outf = clp.getOutputFiles();
            Logger.info ("Output files:      "  + CLP.arrayToString(outf));

            Parameter param = cfm.getParameter ("los.LOS.iDomain");
            Double[] valueVectorDouble = param.getVectorDouble();
            for (int i = 0; i < valueVectorDouble.length; i++) {
                System.out.println(valueVectorDouble[i]);
            }

            param = cfm.getParameter ("matrix5x4");
            Integer[][] matrix = param.getMatrixInt();
            for (int i = 0; i < matrix.length; i++) {
                for(int j = 0; j< matrix[i].length; j++) {
                    System.out.println(matrix[i][j]);
                }
            }
        } catch (Exception e) {
            Logger.error("TestModule failed");
            Logger.finishExecution(1);
        }
    }
} // End of main
```

4.4.8.1. Java Compilation and Execution process.

Use of OSFI support for Java modules requires Java 7 or higher.

In order to use OSFI in Java the environment variable JAVA_HOME needs to be configured to point to the Java virtual machine to consider.

```
export JAVA_HOME= /usr/lib/jvm/java-8-oracle
```

The compilation process needs to specify the base location of the OSFI packages with these environment variables:

- OSFI_HOME, typically < openSF_install_dir>/OSFI

To compile your sources you must specify the location of the osfi jar library. This command is a valid example for compiling the above provided example.

```
javac -cp $OSFI_HOME/lib/osfi.jar TestModel.java
```

The command line for executing the example is:

```
java -cp .:$OSFI_HOME/lib/osfi.jar TestModel <arguments>
```

For execution from openSF the user module needs to packaged as a jar file:

```
jar -cvmf MANIFEST.MF TestModel.jar TestModel.class
```

where the manifest file can be filled as shown below

```
Manifest-Version: 1.1
Created-By: Deimos Space, SL
Main-Class: TestModel
Class-Path: lib/osfi.jar
```

End of Document