EE XML/Binary CFI File Handling User Manual

Code : SO-UM-DME-L1PP-0005
Issue : 2.5
Date : 02/12/08

	Name	Function	Signature
Prepared by	N. Almeida	Project Engineer	
	M. Ventura	Project Engineer	
	A. Lopes	Project Engineer	
Checked by	J. Freitas	Quality A. Manager	
Approved by	J. Barbosa	Project Manager	

DEIMOS Engenharia
Av. D. João II, Lote 1.17, Torre Zen, 10º
1998-023 Lisboa, PORTUGAL
Tel: +351 21 893 3017
Fax: +351 21 896 9099
E-mail: mailto:deimos@deimos.com.pt

This page intentionally left blank

Document Information

Contract Data		Classification	
Contract Number:	DE04/B-434/P	**Internal**	☐
		Public	☐
Contract Issuer:	EADS CASA Espacio	**Industry**	☐
		Confidential	☑

Internal Distribution		
Name	Unit	Copies

External Distribution		
Name	Organisation	Copies
Josep Closa	EADS CASA Espacio	1
Michele Zundo	ESA	1

Archiving	
Word Processor:	MS Word 2000
File Name:	SO-UM-DME-L1PP-0005-BINXML-FH-SUM
Archive Code:	P/SUM/DME/03/013-036

Document Status Log

Issue	Change description	Date	Approved
1.0	Table of Contents	2004-10-30	
1.1	Delivery version for ADR	2004-12-20	
1.2	Delivery after reviews from ADR	2005-02-21	
1.3	Delivery with the release of library	2005-02-23	
1.4	F.A.T. v0	2005-03-30	
1.5	Release of library version 1.3	2005-05-02	
1.6	Updates after CDR RIDs	2005-08-31	
1.7	Updates for version 1.4 of library, including variable arrays	2005-09-23	
1.8	Updated for version 3.0 of the library. Data Set offset starts from the beginning of the binary part of the product	2006-07-20	
2.0	Updated for version 3.2 of the library. BinX dependency was removed and dependencies updated	2006-11-17	
2.1	Description of Nested Variable Arrays in Future Work section.	2007-03-28	
2.2	Description of Nested Variable Arrays	2007-11-28	
2.3	Updated for version 3.5 of the library. Modifications in the support of nested variable arrays	2008-04-18	
2.4	Updated for version 3.6 of the library. Updated descriptions of ‘prepare_schema_nested’.	2008-07-22	
2.5	Updated for version 3.7 of the library. Added information about new function “get_fixed_array” and about “test_nested”.	2008-12-02	

Table of Contents

1. INTRODUCTION 1
1.1. Purpose and Scope 1
1.2. Acronyms and Abbreviations 1
1.3. Applicable and Reference Documents 1
1.3.1. Applicable Documents 1
1.3.2. Reference Documents 2
2. LIBRARY GUIDE 3
2.1. Objectives 3
2.2. Components 3
2.2.1. Object Handler 3
2.2.2. API 4
2.3. Compile and link 5
2.4. Usage 6
2.5. Configuration File 9
2.6. Hybrid file header 10
2.7. Template file 11
2.8. Library Self Test 11
2.9. Constraints 12
2.10. Degree of Portability 12
3. BRIEF DESCRIPTION 14
3.1. Defines 14
3.2. Enumerated Values 14
3.3. Structures/ Types 14
3.4. Function List 16
4. DETAILED DESCRIPTION 18
4.1. Defines 18
4.2. Enumerated Values 18
4.2.1. eErrorCode - Error Codes 18
4.2.2. eErrorLevel - Error Level 19

4.2.3. eStartBit – Start Bit 19
4.3. Structures / Types 19
4.4. Function Description 21
4.4.1. Initializing and Terminating functions 21
4.4.2. Retrieving Functions 27
4.4.3. Storing Functions 36
5. ANNEX 1 – DISK DATA ACCESS 44
6. ANNEX 2 – BINX OBJECT DESCRIPTION 45
6.1. Used Types 45
6.2. Used Methods 45
7. ANNEX 3 – EARTH EXPLORER File Handling CFI 47
7.1. Methods Used 47
8. ANNEX 4 – Self test cases 48
8.1. test_binxml 48
8.2. test_nested 51
8.3. file_transform 52
9. ANNEX 5 – Nested Variable Arrays 53
9.1. Purpose 53
9.2. Limitations 56
9.3. Approach 56

List of Figures

Figure 1: Accessing data variables. 4
Figure 2: Directory structure of the library. 6
Figure 3: Retrieving a scene from a hybrid XML data file. 7
Figure 4: Example of how to use the Library to retrieve information. 8
Figure 5: An example of how to use the Library to save information. 9
Figure 6: BinX data structure. 45
Figure 7: Retrieving data from a hybrid XML data file using the variable array approach. 55
Figure 8: Generating the binary file using variable array approach. 55

List of Tables

Table 1: Table of Acronyms. 1
Table 2: Applicable Documents. 1
Table 3: Reference Documents 2
Table 4: Library dependencies 5
Table 5: Time used accessing a variable of N scenes (in seconds). 44

1. INTRODUCTION

1.1. Purpose and Scope

This document describes the EE XML/Binary CFI File Handling Library for accessing (read/write) binary files. This CFI is to be used initially in the development of the SMOS L1 Processor Prototype in order to read already created binary files and write the different level products of the L1 Processor. Detailed descriptions of each of the functions provided by the CFI are described here for reference.

The library is developed in such a way that it is independent of the SMOS mission, being possible to use it for any kind of XML/Binary files that follow some specific guidelines. The functions presented in this document shall not make any explicit reference or assumptions related exclusively to the SMOS L1 Prototype.

1.2. Acronyms and Abbreviations

API	Application Programming Interface
CFI	Customer Furnished Item
Hybrid-XML	A file with a header written in XML concatenated with the binary information
EEFH	Earth Explorer File Handling CFI (ASCII XML library)
LSB	Least Significant Bit
MSB	Most Significant Bit
EE	Earth Explorer

Table 1: Table of Acronyms.

For the complete list of acronyms, please refer to the document SO-LI-CASA-PLM-0094 “Directory of Acronyms and abbreviations”.

1.3. Applicable and Reference Documents

1.3.1. Applicable Documents

Ref.	Code	Title	Issue
SOW	SO-SOW-CASA-PLM-0855	Level 1 Processor Prototype Development Phase 3 and Support and Analysis Activities. Statement of Work	***01***
BinX	BinX	Editkt::BinX 1.2 Developer’s Guide	***1.2***

Table 2: Applicable Documents.

1.3.2. Reference Documents

Ref.	Code	Title	Issue
RD 1	CS-MA-DMS-GS-0001	Earth Explorer Mission CFI Software MISSION CONVENTIONS DOCUMENT	1.3
RD 2	PE-TN-ESA-GS-0001	Earth Explorer Ground Segment File Format Standard	1.4
RD 3	CS-MA-DMS-GS-0002	Earth Explorer Mission CFI Software GENERAL SOFTWARE USER MANUAL	3.7
RD 4	CS-MA-DMS-GS-0008	EXPLORER FILE HANDLING Reference Manual	3.7
RD 5	SO-LI-CASA-PLM-0094	Directory of Acronyms and abbreviations	

Table 3: Reference Documents

2. LIBRARY GUIDE

2.1. Objectives

The main objective of this CFI is to provide access to the data of a binary file using a XML Schema that defines the order and type of the data contained in that binary file. This approach makes it possible to alter the read/write format of a given binary file by simply changing the XML Schema. The CFI shall also provide access capabilities to hybrid XML/Binary files, where the binary file has a XML ASCII header preceding it. The binary content in such files shall not be enclosed within tags.

The CFI is based on BinX (http://www.edikt.org/binx/download/index.jsp), which provides the storage and retrieval of data into binary files, and is also based on the Earth Explorer File Handling library for storage and retrieval of data into XML files.

The CFI shall provide all the mechanisms internally to separate the XML and the binary parts for separate access, making it transparent to the user. Each part will need an independent validation schema, describing the format and type of the values to be read from the XML and binary contents. The BinX based schema and functions provide a direct conversion from the binary part contents into memory variables, whose type is defined in the schema.

The intended use of this library is for the decoding of hybrid binary-XML product files within the SMOS L1 Processor Prototype. This library shall be the only interface for handling read and write functions within the prototype. Handling of pure XML product files shall be covered by the already available EE File Handling CFI.

2.2. Components

The library is separated in two different components: the component that deals with objects and that is related with the fact that the BinX library is written in C++, i.e., the Object Handler; and the component that provides the functions that are available for the user in ANSI C, i.e., the API.

2.2.1. Object Handler

The Object Handler is implemented in C++ and linked with the API. The main goal of this handler is to provide to the API an efficient way to handle binary data files using XML-Schemas. To respond to this demand the Object Handler uses the BinX library that makes possible to load in memory the content of binary files or writing data into them. These two operations are done based on a BinX language file that describes the binary data structure and allows BinX to translate it into an XML-Schema.

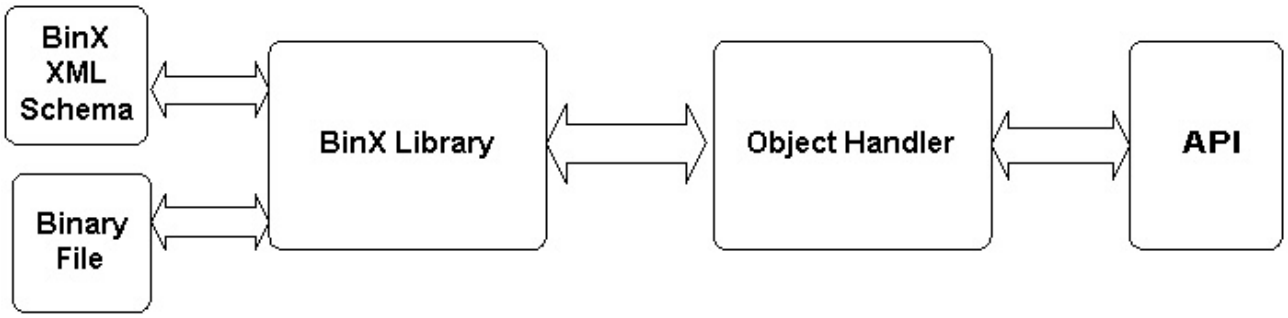

Figure 1: Accessing data variables.

The Object Handler uses the BinX API functions to load the data into memory, thus creating objects that represent that information. When the objects are created, a small table identifying the new objects is filled with the object and an object identifier (an integer). By this approach it shall be very easy for the API to know the available objects and even to destroy those that are no longer need.

The Object Handler management table will have the following structure:

ObjectID	Object
`int` ID1	* object1

The ObjectID field identifies in a unique way a memory loaded object.

To describe better the role of Object Handler let us consider that binary data file with several structures is going to be loaded in memory using BinX. Imagine that the user wants at least one variable of each structure. To solve this problem the CFI builds a BinX Schema file describing the structures the user wants and then uses the Object Handler to load all the structures required into memory.

When a structure is loaded a new entry is added to the table associating an identifier with that structure (the Object). Then, the user requests to the API the structure's variable providing the identifier and the variable name. The Object Handler then returns to the API the value of the required variable.

The Object Handler is also responsible for returning to the API the structure content of an object, when requested, i.e., given a structure, the Object Handler must fill that structure, retrieving the information from the object that represent the information loaded from the binary data file using BinX.

The library imposes no limits in the amount of data committed to memory, although the user must be aware that loading more data than the available RAM shall most probably result in loss of performances.

2.2.2. API

The API is implemented in ISO C99, and represents the functions that are available for the user. It shall return the values gathered from the Object Handler to the user.

2.3. Compile and link

The library has the following dependencies:

Compilation Library	Version	Library Name
libxml2	2.6.22 or newer	XML C parser and toolkit
libexplorer_file_handling	3.7	Earth Explorer File Handling
libxerces-c	2.7.0 or newer	Apache Xerces XML parser

Table 4: Library dependencies

Since the BinX library was discontinued, the dependency from the BinX was removed. Instead, BinXML-FH now includes a modified version of BinX library – BinX 1.2.4 – which is not an official release and is only used internally by BinXml-FH.

The libxml2 library is required by the CFI library. The Xerces library is required by BinX. For more details on how to install each one of the libraries please refer to each library's documentation.

The dependencies of the current library are only on the EE File Handling CFI and on the BinX library. These previous libraries, and not the library described in this document, impose the dependency on libxml and libxerces. Tests have been made with more advanced versions (e.g. libxml 2.6.31) without any problem, the shown recommended versions are taken from the BinX and EE CFI documentation.

The library is structured as shown in Figure 2, where the `src` directory contains the source files (*.c/*.cc), the `include` directory contains the headers (*.h) required to compile the library's components and the `obj` directory contains the objects resulting from the compilation of the sources. The library will be created in the `lib` directory.

The `test` directory contains a program that can be used to check if the library is properly installed. There are some templates in the directory `schema-templates` and one hybrid file in `test/data` that are used by this test program. For more information on how to run it, please see Library Self Test.

```
binxml-fh
    include
        api
        common
        object-handler
    lib
    obj
        api
        common
        object-handler
    schema-templates
    src
        api
        common
        object-handler
    test
        bin
        obj
        src
```

Figure 2: Directory structure of the library.

To compile the library it is necessary to use the `Makefile` that exists on the root of the library. There is another dedicated `Makefile` in the test directory, which shall be used to generate the self-validation test. This validation test can also be used as an example of how the API functions shall be used.

For the latest information on how to install the library, please check the file `readme.txt` located at the root directory

2.4. Usage

The library has its own configuration file (`binxml-fh.conf.xml`) that allows configuring some required parameter of the library [see 2.5 Configuration File].

It is necessary to give permission to the user of the library to write files and to give him quota enough to deal with the temporary files created while using the library, either for reading and writing. The worst case estimated quota that it shall be necessary, according to the files that the user is going to process, shall be double the size of the processed files, to allow complete rewriting of each file into a temporary directory.

For accessing values defined in the ASCII part of the hybrid XML (the header) the libray shall use the EEFH [RD 4], as already defined, providing access to it to the user as well through the same interface. The library shall also provide access to the Binary part of the hybrid XML file, through functions described in subsequent chapters.

To explain how the library works, let us consider a SMOS L1 applicable scenario, where it exists a block of data called Scene and a hybrid-xml file (L0 Product) containing a specific number of scenes. The header of the hybrid-xml contains a tag with the number of scenes within the file and another tag explaining the offset where the binary data starts in the hybrid-xml file.

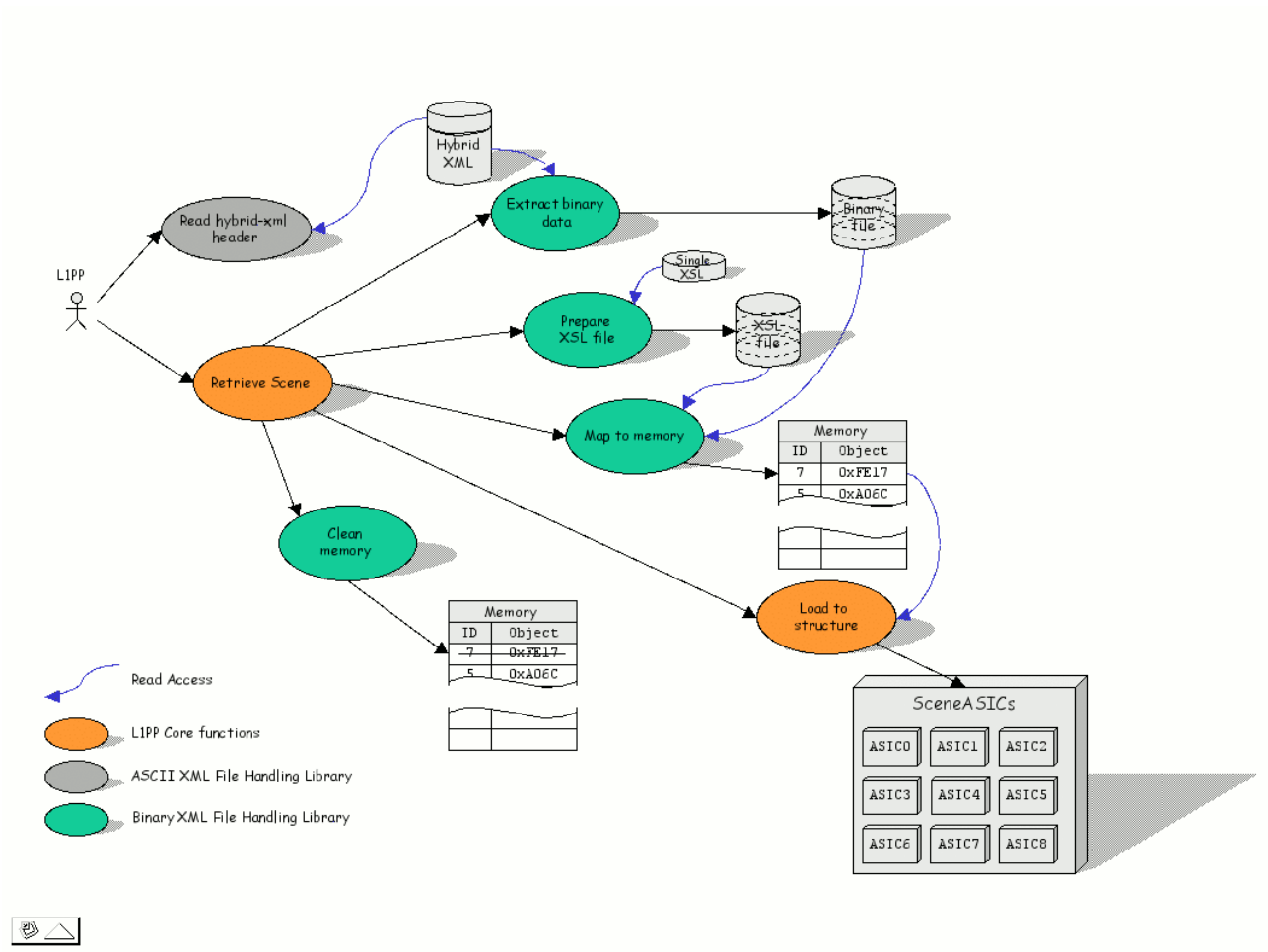

Figure 3: Retrieving a scene from a hybrid XML data file.

As it is shown in Figure 3, the CFI starts by accessing the header of the hybrid-xml file to know how many scenes exists in the file and provides this information to the user, who then requests the retrieval of a specific scene (or an array of scenes). To retrieve the required scene, the user must implement a function that uses the API of the library to retrieve the information and fill the user's data structure. For that purpose, the user calls the function of the API – `extract_data()` – to extract the desired block of data (scene) from the hybrid-xml file and write it into a binary data file. Then the user will request the BinXML-FH to prepare the schema that characterizes the block data (scene) that was extracted previously – `prepare_schema()`. With the binary file and the schema ready, the user requests the BinXML-FH to map into memory the binary file using the schema – `load_data_file()`. This returns an identifier that has to be used to retrieve the variables' value in order to fill the user defined structure (Load to structure). This is done calling the proper `get_...()` functions, as required by the structure. After the values of the variables have been inserted in the user defined structure, the user can request the library to release the memory allocated to the binary file using the definition of the schema using the identifier that was returned previously – `free_data_file()`.

After this process, the user has stored the requested scene in its own defined structure, using the memory allocated by him. In the following image, the previous process is shown, also providing a user defined function `get_scene()`, which includes all the CFI function calls.

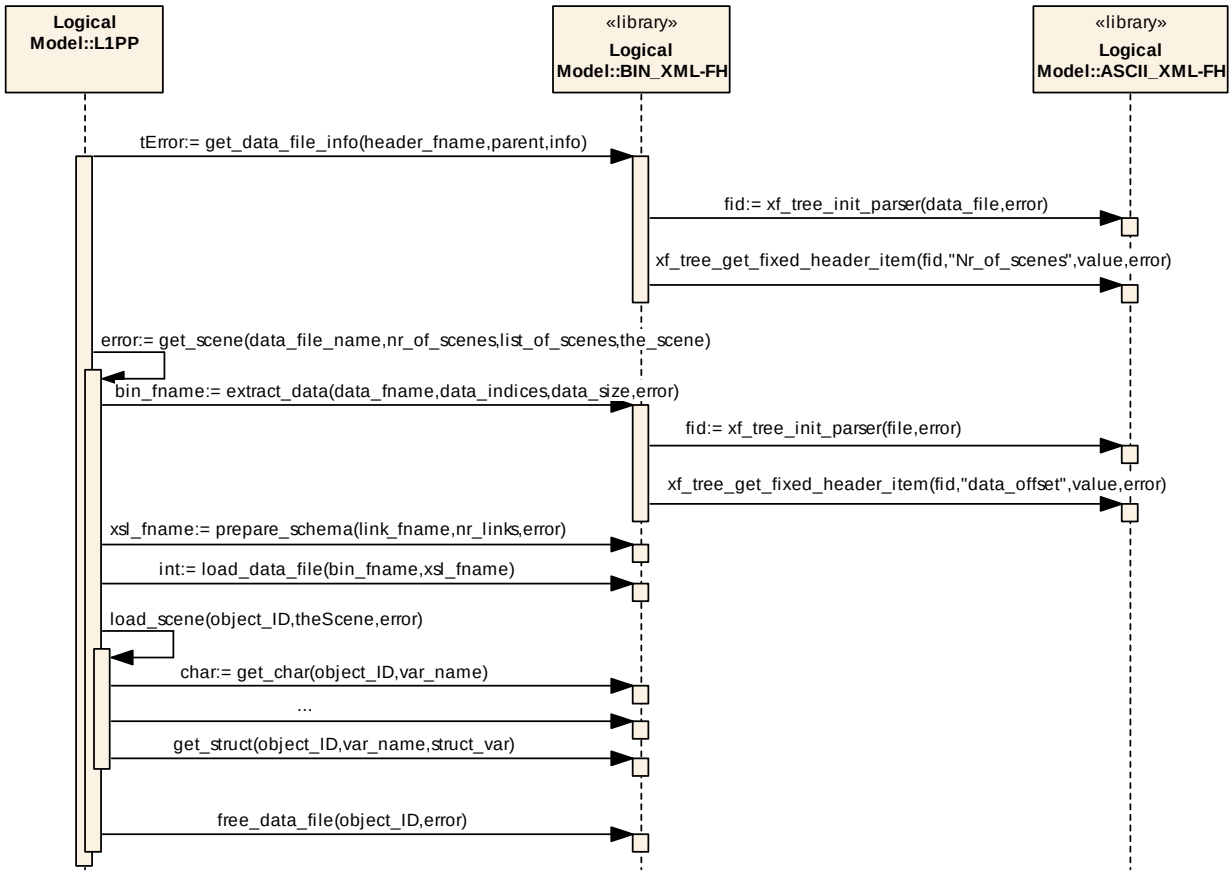

Figure 4: Example of how to use the Library to retrieve information.

The Figure 4 shows in more detail the sequence of the functions of the BinXML-FH that are called to implement the user function that retrieves the data block (Scene) and fills the user defined structure, as described previously. In this figure the EEFH is identified as ASCII_XML-FH.

To perform the reverse process, store the data block, it is necessary to prepare the schema (or use an existing one if that is the case) that describes the way the information is to be stored – `prepare_schema()` – and then create a representation of the binary file in memory using that schema – `load_data_file()`. This returns an identifier that is used to pass the information to the API using the `set_…()` functions. After all the information have been passed, the file can be written to disk – `save_data_file()`.

An example of the saving process is shown in Figure 5, where a user defined function `save_scene()` is also represented, grouping the CFI use in a single function.

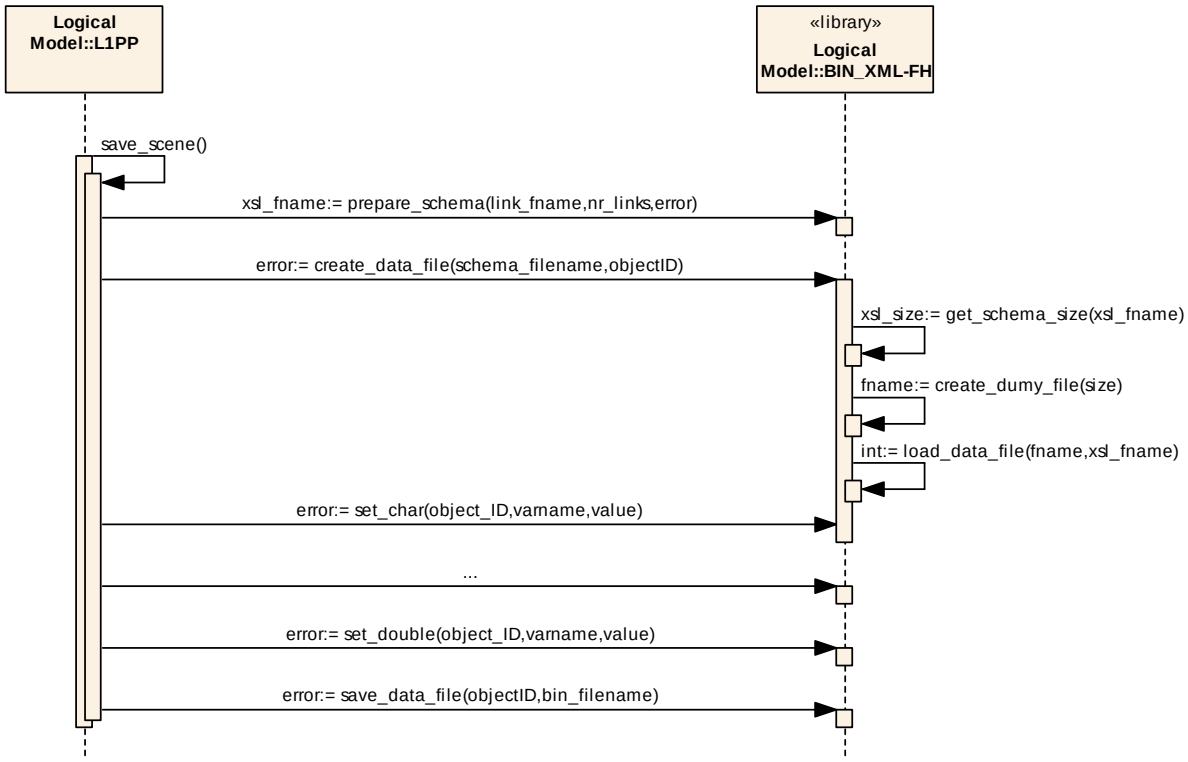

Figure 5: An example of how to use the Library to save information.

2.5. Configuration File

Before using the functions provided by the library it is necessary to read the configuration file using the function `read_config_file()`.

The structure of the configuration file is as follows:

```
<?xml version="1.0"?>
<config>
  <lib_home_dir>/opt/binxml-fh/</lib_home_dir>
  <tmp_data_dir>../tmp/data/</tmp_data_dir>
  <dataset_string>dataset src</dataset_string>
  <header_size_string>HEADER_SIZE</header_size_string>
  <top_lines_of_header>25</top_lines_of_header>
  <tag_block_list>DSD_LIST</tag_block_list>
  <tag_block_set>DATA_SET</tag_block_set>
  <tag_block_set_name>DATA_SET_NAME</tag_block_set_name>
  <tag_block_set_size>MDR_SIZE</tag_block_set_size>
  <tag_block_set_count>NUM_MDR</tag_block_set_count>
  <tag_block_set_offset>MDR_OFFSET</tag_block_set_offset>
  <tag_byte_order>Byte_Order</tag_byte_order >
</config>
```

The tag `lib_home_dir` identifies the library path and is used by the function `read_config_file()` when the configuration file is passed to it as argument.

The tag `tmp_data_dir` indicates the directory where the temporary files needed by the library will be created. These temporary files are the binary files and the schema files.

The tag `dataset_string` specifies a string that contains the tag and attribute of the template, where it is indicated the variable type and name. [see 2.6 Hybrid file header]

The tag `header_size_string` specifies the tag from the header of the hybrid-XML file that indicates the size of that same header.

The tag `top_lines_of_header` indicates the number of lines from the header of the hybrid-XML file that will be read looking for the string defined by the tag `header_size_string`.

Because the hybrid file may contain different types of data, here referred as blocks, the header of the hybrid file must contain a list with a specific number of blocks. Each block must contain a name, the offset to the data block starting from the beginning of the hybrid file, the number of elements that are in the mentioned block and the size of each on of its elements.

The tags of the configuration file, that define the tags of the block list contained in the header of the hybrid file, are the following. The `tag_block_list` defines the tag that identifies the beginning of the list of blocks. This tag should contain an attribute `count` that specifies the number of blocks contained in the hybrid file. The start of a block description is identified by the tag defined in `tag_block_set`. The block name and offset are specified by the tags defined in `tag_block_set_name` and `tag_block_set_offset` respectively. The offset of each data block is always related to the start of the file. The number of elements contained in each block is specified by the tag defined in `tag_block_set_count` and the size of each element is identified in the tag defined by `tag_block_set_size`.

The tag `tag_byte_order` is used to identify the byte order used in the data block.

2.6. Hybrid file header

Here is a description of the required elements of the hybrid-XML header.

The header of the hybrid file must contain at the beginning the tag specifying the size of the header. This tag is defined in the configuration file by the tag `header_size_string` and must be in the first lines as defined by `top_lines_of_header` in the configuration file. The header must also contain a block list with the tags defined in the configuration file.

In the following paragraph there is an example of such a header:

```
<FILE>
   <HEADER_SIZE>0000005367</HEADER_SIZE>
   <DSD_LIST count="01">
      <DATA_SET>
         <DATA_SET_NAME>CHARACTERS                    </DATA_SET_NAME>
         <MDR_SIZE>+0000002</MDR_SIZE>
         <NUM_MDR>+00075</NUM_MDR>
         <MDR_OFFSET>+0005367</MDR_OFFSET>
         <Byte_Order>3210</Byte_Order>
      </DATA_SET>
   </DSD_LIST>
</FILE>
```

2.7. Template file

The template file is used to generate the schemas required by BinX to read the binary file.

It is a XML schema file following the specification of BinX [BinX] but with some restrictions:

- in the schema template, after the tag defined by `dataset_string` in the configuration file, only the first line containing the `useType` tag will be taken into account when creating the schema using the `prepare_schema` function. Any tag and/or comment after this line is ignored.
- the variable name must contain the # character that identifies the place where the index of the element will be placed.

Following it is an example of a template:

```
<?xml version="1.0" encoding="UTF-8"?>
<binx xmlns="http://www.edikt.org/binx/2003/06/binx">

   <definitions>
        <defineType typeName="Basic_Types">
           <struct>
             <unsignedInteger-32 varName="var_integer" />
             <float-32 varName="var_float" />
        <double-64 varName="var_double" />
             <unsignedLong-64 varName="var_long" />
        <unsignedByte-8 varName="var_char" />
           </struct>
       </defineType>
   </definitions>

<dataset src="binary_product" byteOrder="bigEndian">
     <useType typeName="Basic_Types" varName="B_Types[#]" />
</dataset>
</binx>
```

2.8. Library Self Test

To compile the test, it is necessary to compile the BinXML library first and include the library directory on the `LD_LIBRARY_PATH`, or (`DYLD_LIBRARY_PATH` for MacOS users). Then, in the test directory, run the `make` command to build the test program.

```
make test_binxml
```

The syntax of the program is the following:

```
test_binxml
```

To run the test it is simply required to invoke the name of the executable. Then the test can be run in two different ways:

1) call directly the tester passing as argument the configuration file

```
bin/test_binxml binxml_config.xml
```

2) define the environment variable `BINXML_HOME` as the home directory of the library and than run the test without specifying the configuration file. In this case the library will be using the configuration file in its home directory as defined by the environment variable. It is important to include the slash '/' at the end of the path.

```
export BINXML_HOME=/opt/binxml-fh/
bin/test_binxml ../test/data/hybrid_binxml_test.eef ../schema-templates/doublecomplex_template.xml
```

The first time that it is run the configuration file does not exists, so it is created and it is necessary to restart the test. The test shall write known values into a Hybrid file using the write functions, and it shall later open it and decode all the variables, comparing them to the expected values.

An output like the one presented in Annex 4 shall be shown, identifying the status of every function that can be used from the API. If everything is OK and no error is shown, the library is correctly installed and ready to use.

2.9. Constraints

As mentioned before, the provision of the binary format in a schema allows altering such format directly into the schema without the need of recompilation of the tool using the CFI. However, the type of data of a particular variable contained in the binary file cannot change drastically from the original type, so that the type used in the implementation still makes sense. For example, if a variable is written in the binary file as a char and afterward an integer is used to represent that same variable, the library will not be able to return the correct value, because surely the tool implementation using the CFI shall be expecting a char, and the representation of a char is up to 255.

This feature is present in this version 1.3 of the library, which does not return any error if the user tries to load a variable of one type (i.e. double) with a different function (i.e. get_int). The library automatically assigns the proper values so the memory is not corrupted. In the current version, a warning is returned for any “set” function each time the user does not use the appropriate function for the type of variable being used. Additionally, “get” functions inform the user when the variable retrieved exceeds the size of the returning function (i.e. get_int used to retrieve double valued variable).

In order to decode a hybrid-xml file, the CFI needs to know how to separate the binary component from the XML header to apply separate schemas. This is done by forcing the definition of the tag “HEADER_SIZE” somewhere in the XML header, containing the size in bytes of the XML header component. This value shall be read by the CFI using default C reading functions, as it shall not be possible to apply XML parse functions until the header is separated. [see 2.6 Hybrid file header]

This characteristic is needed to be able to read hybrid-files whose binary component is not enclosed within tags. XML parse functions are not able to decode a file whose complete content is not enclosed in tags, and defining a binary part within an XML file could cause a parsing error (although highly improbable) as the ASCII representation of the binary characters could match a closing tag.

It is necessary as well to reserve some special characters for defining the BinX schema, as in a list of consecutive similar data blocks the type is common to all of them, but the value name shall change from one to the next one. This has been treated in the same way as C arrays, reserving the “[]” characters for it, and the “.” character for concatenation. In this way, a user wanting to access the NAME value in three consecutive SCENE data blocks would need to ask the CFI for the following values: “SCENE[1].NAME”, “SCENE[2].NAME”, “SCENE[3].NAME”. [see 2.7 Template file]

2.10. Degree of Portability

The library has been tested on the following environments:

- SUSE 8
- RedHat Fedora

- SUN Solaris 5.8
- MacOS X Darwin 7.9.0

More OS shall be added as the multi-platform testing proceeds.

3. BRIEF DESCRIPTION

The following chapter provides a quick reference guide to definitions, enumerated values, structures, error codes and function interfaces. For a detailed guide, the next chapter shall be used.

3.1. Defines

```
#define CONFIG_FILE “binxml-fh.conf.xml”
#define MAX_FILE_NAME_SIZE    256
#define MAX_ERROR_MSG_SIZE    256
#define MAX_LINE_SIZE         256   // Max size of a line read from an ASCII file.
#define MAX_XPATH_SIZE        512   // used to define the Max size of a XPath used in XML.
```

3.2. Enumerated Values

```
enum {
   ERR_CODE_NO_ERROR,
   ERR_CODE_ERROR_ACCESSING_OBJECT,
   ERR_CODE_ERROR_ALLOCATING_MEMORY,
   ERR_CODE_ERROR_FINDING_ENV_VAR,
   ERR_CODE_ERROR_FINDING_STRING,
   ERR_CODE_ERROR_READING_FILE,
   ERR_CODE_ERROR_REMOVING_OBJECT,
   ERR_CODE_TERMINATE,
   ERR_CODE_ERROR_WRITING_FILE,
   ERR_CODE_XML_CFI,
   ERR_CODE_INVALID_OFFSET,
   ERR_CODE_INVALID_DATA_TYPE
} eErrorCode;

enum {
   ERR_LEV_LOW,
   ERR_LEV_MEDIUM,
   ERR_LEV_HIGH
} eErrorLevel;

enum {
   MSB,
   LSB
} eStartBit;
```

3.3. Structures/ Types

```
typedef eStartBit tStartBit;

typedef eErrorCode tErrorCode;

typedef eErrorLevel tErrorLevel;

typedef struct _Error{
   tErrorCode ErrorCode;
   tErrorLevel ErrorLevel;
   char ErrorMsg[MAX_ERROR_MSG_SIZE];
} tError;

typedef struct _DataBlockInfo {
```

```
   char *name;
   long offset;
   long element_size;
   long count;
   char endian;
} tDataBlockInfo;

typedef struct _Indices {
   int count;
   int *value;
} tIndices;

typedef struct _complex{
   float r;
   float i;
} tComplex;

typedef struct _doublecomplex{
   double r;
   double i;
} tDoubleComplex;

typedef struct _Vector{
   void *value;
   unsigned long count;
}tVector;

typedef struct _VectorChar{
   char *value;
   unsigned long count;
}tVectorChar;

typedef struct _VectorInt{
   int *value;
   unsigned long count;
}tVectorInt;

typedef struct _VectorLong{
   long long *value;
   unsigned long count;
}tVectorLong;

typedef struct _VectorFloat{
   float *value;
   unsigned long count;
}tVectorFloat;

typedef struct _VectorDouble{
   double *value;
   unsigned long count;
}tVectorDouble;

typedef struct _VectorShort{
        short* value;
        unsigned long count;
}tVectorShort;

typedef struct _VectorComplex{
        tComplex* value;
        unsigned long count;
}tVectorComplex;

typedef struct _VectorDoublecomplex{
        tDoubleComplex* value;
        unsigned long count;
}tVectorDoublecomplex;

typedef struct _MatrixInt{
        int** value;
```

```
        unsigned long rcount;
        unsigned long ccount;
}tMatrixInt;

typedef struct _MatrixByte{
        char** value;
        unsigned long rcount;
        unsigned long ccount;
}tMatrixByte;

typedef struct _MatrixFloat{
        float** value;
        unsigned long rcount;
        unsigned long ccount;
}tMatrixFloat;

typedef struct _MatrixLong{
        long long** value;
        unsigned long rcount;
        unsigned long ccount;
}tMatrixLong;

typedef struct _MatrixDouble{
        double** value;
        unsigned long rcount;
        unsigned long ccount;
}tMatrixDouble;

typedef struct _MatrixDoublecomplex{
        tDoubleComplex** value;
        unsigned long rcount;
        unsigned long ccount;
}tMatrixDoublecomplex;
```

3.4. Function List

```
tError append_file(char *fname_in, char *fname_out);
tError copy_file(char *fname_in, char* fname_out);
tError create_config_file(char *ConfigFileLocation);
tError read_config_file(char *config_file_name);
void free_config_file();
tError join_to_hybrid_file(char* fname_header, char* fname_binary, char *fname_hybrid);
tError split_hybrid_file(char *fname_hybrid);

tError get_bit(int object_ID, char *varname, tStartBit StartBit, int bit_nr, char *value);
tError get_bits(int object_ID, char *varname, int bit_start_nr, int bit_stop_nr, const char
**value);
tError get_char(int object_ID, char *varname, char *value);
tError get_byte(int object_ID, char *varname, char *value);
tError get_str(int object_ID, char *varname, char **value);
tError get_short(int object_ID, char *varname, short *value);
tError get_int(int object_ID, char *varname, int *value);
tError get_long(int object_ID, char *varname, long long *value);
tError get_float(int object_ID, char *varname, float *value);
tError get_double(int object_ID, char *varname, double *value);
tError get_complex(int object_ID, char *varname, tComplex *value);
tError get_doublecomplex(int object_ID, char *varname, tDoubleComplex *value);

tError get_fixed_array(int object_ID, char* varname, tVector* vector);
tError get_variable_array(int object_ID, char* varname, tVector* vector);
tError get_vector_byte(int object_ID, char* varname, tVectorChar* array_char);
tError get_vector_long(int object_ID, char* varname, tVectorLong* v_long);
tError get_vector_int(int object_ID, char* varname, tVectorInt* array_int );
tError get_vector_double(int object_ID, char* varname, tVectorDouble* array_double);
tError get_vector_short(int object_ID, char* varname, tVectorShort* array_short );
tError get_vector_float(int object_ID, char* varname, tVectorFloat* array_float );
```

```
tError get_vector_complex(int object_ID, char* varname, tVectorComplex* array_complex );
tError get_vector_doublecomplex(int object_ID, char* varname, tVectorDoublecomplex* array_dcomplex
);

tError get_matrix_int(int object_ID, char* varname, tMatrixInt* matrix_int );
tError get_matrix_byte(int object_ID, char* varname, tMatrixByte* matrix_byte );
tError get_matrix_float(int object_ID, char* varname, tMatrixFloat* matrix_float );
tError get_matrix_long(int object_ID, char* varname, tMatrixLong* matrix_long );
tError get_matrix_double(int object_ID, char* varname, tMatrixDouble* matrix_double );

tError get_header_size(char* hybrid_fname, int *size);
tError extract_header(char *hybrid_fname, char **hdr_fname, int *header_size);
tError get_data_file_info(char* filename, char *parent, tDataBlockInfo *info);
tError prepare_schema(char *fname_bin, char *fname_link, tIndices var_indices, char **fname_schema,
char endian);
tError prepare_schema_nested(char *fname_bin, char *fname_link, char **fname_schema, char *endian);
tError   extract_data(char   *data_filename,   char   *parent_name,   tIndices   block_indices,char
**bin_filename);
tError load_data_file(char *schema_filename, int *objectID);
tError create_data_file(char *schema_filename, int obj_size, int *objectID);
tError free_data_file(int object_ID);

tError save_data_file(int objectID, char *bin_filename);

tError set_bit(int object_ID, char *varname, tStartBit StartBit, int bit_nr, char value);
tError set_char(int object_ID, char *varname, char value);
tError set_str(int object_ID, char *varname, char* value);
tError set_short(int object_ID, char *varname, short value);
tError set_int(int object_ID, char *varname, int value);
tError set_long(int object_ID, char *varname, long long value);
tError set_float(int object_ID, char *varname, float value);
tError set_double(int object_ID, char *varname, double value);
tError set_complex(int object_ID, char *varname, tComplex value);
tError set_doublecomplex(int object_ID, char *varname, tDoubleComplex value);

tError set_vector_byte(int object_ID, char *varname, tVectorChar v_char);
tError set_vector_short(int object_ID, char *varname, tVectorShort v_short);
tError set_vector_int(int object_ID, char *varname, tVectorInt v_int);
tError set_vector_long(int object_ID, char *varname, tVectorLong v_long);
tError set_vector_float(int object_ID, char *varname, tVectorFloat v_float);
tError set_vector_double(int object_ID, char *varname, tVectorDouble v_double);
tError set_vector_doublecomplex(int object_ID, char *varname, tVectorDoublecomplex v_doublecomplex);

tError set_matrix_int(int object_ID, char *varname, tMatrixInt matrix_int);
tError set_matrix_float(int object_ID, char *varname, tMatrixFloat matrix_float);
tError set_matrix_long(int object_ID, char *varname, tMatrixLong matrix_long);
tError set_matrix_double(int object_ID, char* varname, tMatrixDouble matrix_double );
tError set_matrix_byte(int object_ID, char* varname, tMatrixByte matrix_byte );
tError set_matrix_doublecomplex(int object_ID, char *varname, tMatrixDoublecomplex matrix_dcomplex);

void* readNestedData(char *xml_fname, void **objectID);
char* getDataFormat(void *objectID);
void writeNestedData(void *data, char *xml_fname, char *fname);
void freeNestedData(void **objectID);
```

4. DETAILED DESCRIPTION

4.1. Defines

- CONFIG_FILE

It defines the name of the configuration file of the library.

- MAX_FILE_NAME_SIZE

It defines the maximum size allowed for identifying a file with the full path.

- MAX_ERROR_MSG_SIZE

It defines the maximum size of the message describing an error as used in `tError`.

- MAX_LINE_SIZE

It defines the maximum size of the string read from an ASCII file. It is used when preparing a schema with the functions `prepare_schema()`and `prepare_schema_nested()`.

- MAX_XPATH_SIZE

It defines the maximum size of a string containing the XPath used to read/write values from an XML file using the EEFH [RD 4].

4.2. Enumerated Values

4.2.1. eErrorCode - Error Codes

It identifies the type of error that has occurred.

- **NO_ERROR**

Error used to inform that no error have occurred during function processing.

- **ERROR_ACCESSING_OBJECT**

This error occurs when the provided `objectID` doesn't exist.

- **ERROR_ALLOCATING_MEMORY**

This error is used to inform that there was a problem allocating memory.

- **ERROR_FINDING_ENV_VAR**

This code is used when the required environment variable couldn't be found, probably because it is not defined.

- **ERROR_FINDING_STRING**

This code is used when a string is no found in a file or inside another string.

- **ERROR_READING_FILE**

Error used to inform that the requested file could not be opened, either because it doesn't exist or it hasn't the proper permissions.

- **ERROR_REMOVING_OBJECT**

This error occurs when the provided `objectID` doesn't exist.

- **TERMINATE**

This code is used to inform that a termination or re-initialization of the library is required, for example, when the default configuration file is created using the function `create_config_file`().

- **ERROR_WRITING_FILE**

Error used to inform that the requested file could not be written into, either because it doesn't exist or it hasn't the proper permissions.

4.2.2. eErrorLevel - Error Level

There are 3 levels of errors:

1. ERR_LEV_LOW, is the lowest of all and usually is associated with errors that doesn't affect the library too much. It works like a warning.
2. ERR_LEV_MEDIUM, is the intermediate level. The program can follow, but some action must be taken in order to not break the program.
3. ERR_LEV_HIGH, is the highest level and it usually identifies an error that will require the termination the program or the reloading of the library's configurations.

4.2.3. eStartBit - Start Bit

It can be MSB or LSB and is used to identify the starting bit of the counting.

Example:

Considering the byte 10101110 and that the left bit is the more significant bit, the get_bit(…, MSB,1,…) will return the 2nd bit counting from left (0) while get_bit(…, LSB,1,…) will return the 2nd bit counting from right (1).

4.3. Structures / Types

- **tError**

Structure used to contain the error information.

Composed by ErrorCode and ErrorLevel enumerated plus an array of characters to incorporate additional error information.

- **tDataBlockInfo**

Structure containing the information of any Measurement Data Set Descriptor in an XML ASCII header of any hybrid product file. It is used to divide the product file in data blocks for advanced access.

- **tComplex**

This structure is used to represent the complex values composed by two `float` values.

- **tDoubleComplex**

This structure is used to represent the complex values composed by two `double` values.

- **tVector**

This structure is used to represent a one-dimensional array, of unknown type, with the additional information of the number of elements contained in it.

- **tVectorChar**

This structure is used to represent a one-dimensional array of `char` with the additional information of the number of elements contained in that array.

- **tVectorLong**

This structure is used to represent a one-dimensional array of `long long` (C99 Standard) with the additional information of the number of elements contained in that array.

- **tVectorFloat**

This structure is used to represent a one-dimensional array of `float` with the additional information of the number of elements contained in that array.

- **tVectorDouble**

This structure is used to represent a one-dimensional array of `double` with the additional information of the number of elements contained in that array.

- **tVectorInt**

This structure is used to represent a one-dimensional array of `int` with the additional information of the number of elements contained in that array.

- **tVectorShort**

This structure is used to represent a one-dimensional array of `short` with the additional information of the number of elements contained in that array.

- **tVectorComplex**

This structure is used to represent a one-dimensional array of `complex` with the additional information of the number of elements contained in that array.

- **tVectorDoublecomplex**

This structure is used to represent a one-dimensional array of `doublecomplex` with the additional information of the number of elements contained in that array.

- **tMatrixInt**

This structure is used to represent a two-dimensional array of `int` with the additional information of the number of elements contained in that array.

- **tMatrixByte**

This structure is used to represent a two-dimensional array of `byte` values with the additional information of the number of elements contained in that array.

- **tMatrixFloat**

This structure is used to represent a two-dimensional array of `float` values with the additional information of the number of elements contained in that array.

- **tMatrixLong**

This structure is used to represent a two-dimensional array of `long` with the additional information of the number of elements contained in that array.

- **tMatrixDouble**

This structure is used to represent a two-dimensional array of `double` with the additional information of the number of elements contained in that array.

- **tMatrixComplex**

This structure is used to represent a two-dimensional array of `complex` with the additional information of the number of elements contained in that array.

- **tMatrixDoublecomplex**

This structure is used to represent a two-dimensional array of `doublecomplex` with the additional information of the number of elements contained in that array.

- **tIndices**

This is the same as `tVectorInt`, only it used to be more meaningful.

4.4. Function Description

4.4.1. Initializing and Terminating functions

4.4.1.1. create_config_file

Syntax:
```
tError create_config_file(char *ConfigFileLocation);
```

This function creates a default configuration file. There are two possibilities for the location of that file:

1) if the `ConfigFileLocation` is passed as `NULL`, it will be required that the environment variable `BINXML_HOME` is set to the home path of this library with the ending slash '/', for example `'BINXML_HOME=/opt/binxml/'`. In this case, the configuration file will be created in the library home directory.
2) if the `ConfigFileLocation` is not `NULL`, the file will be created in the specified directory.

The function returns a `tError` structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_WRITING_FILE
- ERR_CODE_NO_ERROR

4.4.1.2. read_config_file

Syntax:
```
tError read_config_file(char *config_file_name);
```

This function reads the library's configuration file. It follows the same approach as the function `create_config_file()`: if the `config_file_name` is `NULL` the configuration file will be read from the library's home directory, otherwise the given file name will be read.

The function returns a t`Error` structure to inform the status of the operation.

NOTE: this function allocates memory that is necessary to release at the end of the program or when new configurations have to be loaded - free_config_file().

The errors returned are:

- ERR_CODE_ERROR_FINDING_ENV_VAR
- ERR_CODE_TERMINATE
- ERR_CODE_ERROR_READING_FILE
- ERR_CODE_NO_ERROR

4.4.1.3. free_config_file

Syntax:

```
void free_config_file();
```

This function is used to release the memory allocated to the constant values read from the configuration file. It is usually the last function to be called in the program and also when a new configuration file is to be read.

4.4.1.4. join_to_hybrid_file

Syntax:

```
tError join_to_hybrid_file(char* fname_header, char* fname_binary, char *fname_hybrid);
```

This function is used to join a header and a binary data file into a hybrid-XML file. The header file name and the binary data file name should be the same as the hybrid file name, only the extensions should be different: `eef` for the hybrid file; `hdr` for the header file; and `dbl` for the binary data file.

The function returns a t`Error` structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ALLOCATING_MEMORY
- ERR_CODE_ERROR_READING_FILE
- ERR_CODE_ERROR_WRITING_FILE
- ERR_CODE_NO_ERROR

4.4.1.5. split_hybrid_file

Syntax:

```
tError split_hybrid_file(char *fname_hybrid);
```

This function is used to split a hybrid-XML file into a header file and a binary data file.

The new files are created on the same place that the original one and will have the same name, only the extension will change from `eef` to `hdr` for the header file and `dbl` for the binary data.

The function returns a t`Error` structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ALLOCATING_MEMORY
- ERR_CODE_ERROR_FINDING_STRING
- ERR_CODE_ERROR_READING_FILE
- ERR_CODE_ERROR_WRITING_FILE
- ERR_CODE_NO_ERROR

4.4.1.6. get_header_size

Syntax:

```
tError get_header_size(char* hybrid_fname, int *size);
```

This function returns in the argument `size` the size of the header contained in the hybrid file.

The function returns a t`Error` structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ALLOCATING_MEMORY
- ERR_CODE_ERROR_FINDING_STRING
- ERR_CODE_ERROR_READING_FILE
- ERR_CODE_NO_ERROR

4.4.1.7. extract_header

Syntax:

```
tError extract_header(char *hybrid_fname, char **hdr_fname, int *header_size);
```

This function copies the header of the hybrid-XML file, of name `hybrid_fname`, and stores it on another file whose name is returned in `hdr_fname`.

The function returns a t`Error` structure to inform the status of the operation.

NOTE: This function allocates memory for returning the header file name and the user must release it when it is not used anymore.

The errors returned are:

- ERR_CODE_ERROR_ALLOCATING_MEMORY
- ERR_CODE_ERROR_FINDING_STRING
- ERR_CODE_ERROR_READING_FILE
- ERR_CODE_ERROR_WRITING_FILE
- ERR_CODE_NO_ERROR

4.4.1.8. get_data_file_info

Syntax:

```
tError get_data_file_info(char* filename, char *parent, tDataBlockInfo *info);
```

This function retrieves the description information of a data block described in the "filename" hybrid's file XML ASCII header under tag <DATA> and whose <name> tag is equal to the string "parent" passed as argument.

The tDataBlockInfo structure with the information of the data block is returned to the user in value.

The function returns a tError structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_FINDING_STRING
- ERR_CODE_NO_ERROR

4.4.1.9. extract_data

Syntax:

```
tError extract_data(char *data_filename, char *parent_name, tIndices block_indices,char
**bin_filename);
```

This function extracts one or several data elements (according to block_indices) from a hybrid-XML file of name data_filename, and stores it into a temporary binary file whose name is returned in bin_filename.

The function returns a tError structure to inform the status of the operation.

The data element that is extracted is related with the binary data block identified by parent_name. This section is described in a structure in the header that contains the block name, the offset where the binary data block starts, the size of each element of the block and the number of elements existing in that block. [see 2.6 Hybrid file header]

NOTE: This function allocates memory for returning the binary file name and the user must release it when it is not used anymore.

The errors returned are:

- ERR_CODE_ERROR_ALLOCATING_MEMORY
- ERR_CODE_ERROR_FINDING_STRING
- ERR_CODE_ERROR_READING_FILE
- ERR_CODE_ERROR_WRITING_FILE
- ERR_CODE_NO_ERROR

4.4.1.10. prepare_schema

Syntax:

```
tError prepare_schema(char *fname_bin, char *fname_template, tIndices var_indices, char
**fname_schema, char endian);
```

This function uses a schema template, with file name `fname_template` which describes one element of data, to create another schema containing a specific number of variables of the same type, as defined by the `var_indices` [see 2.6 Hybrid file header]. The new schema is updated specifying the byte order that is used in the binary data (big-endian or little-endian) and is stored in a file whose name is returned in `fname_schema`.

The function returns a t`Error` structure to inform the status of the operation.

NOTE 1: The schemas of BinX require that the binary file is in the same directory of the schema itself, so the file name will be specified in the schema without the full path.

NOTE 2: This function allocates memory for returning the schema file name that the user must release when it is not used anymore.

The errors returned are:

- ERR_CODE_ERROR_READING_FILE
- ERR_CODE_ERROR_WRITING_FILE
- ERR_CODE_NO_ERROR

4.4.1.11. prepare_schema_nested

Syntax:

```
tError prepare_schema_nested(char *fname_bin, char *fname_template, char **fname_schema, char *endian);
```

This function uses a schema template, with file name `fname_template`, describing a nested structure of data, to create another schema that is stored in a file whose name is returned in `fname_schema`. The byte order of the stored binary data (big-endian or little-endian) can be set, passing `*endian == 'b'` or `'l'`, or can be specified in the schema template. In the later case, the user should pass `*endian == '\0'` and the byte order of the template is returned in the same parameter.

The function returns a t`Error` structure to inform the status of the operation.

NOTE 1: The schemas of BinX require that the binary file is in the same directory of the schema itself, so the file name must be specified in the schema without the full path.

NOTE 2: This function allocates memory for returning the schema file name and the user must release it when it is not used anymore.

The errors returned are:

- ERR_CODE_ERROR_READING_FILE
- ERR_CODE_ERROR_WRITING_FILE
- ERR_CODE_NO_ERROR

4.4.1.12. load_data_file

Syntax:

```
tError load_data_file(char *schema_filename, int *objectID);
```

This function loads into memory the data contained in a binary file according with the XML-schema contained in the file with name `schema_filename`. All the information can be accessed by the

variable name associated with the data. It returns an integer `objectID` that identifies the object returned by BinX and that contains the information.

The function returns a t`Error` structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_READING_FILE
- ERR_CODE_NO_ERROR

4.4.1.13. create_data_file

Syntax:
```
tError create_data_file(char *schema_filename, int obj_size, int *objectID);
```

Because BinX does not allow writing into a binary file according to an XML-schema, this function loads a dummy binary file of size `obj_size`, using the schemas stored in `schema_filename`, in order to obtain the identifier – `objectID` – of an object of BinX that respects the data structure defined in the schema. This object can be filled with the required values and then be stored to disk using `save_data_file()`.

The function returns a t`Error` structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_NO_ERROR

4.4.1.14. free_data_file

Syntax:
```
tError free_data_file(int object_ID);
```

This function releases the memory associated with the object identified by `object_ID` and that was previously allocated with `load_data_file()` or with `create_data_file()`.

The function returns a t`Error` structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_REMOVING_OBJECT
- ERR_CODE_NO_ERROR

4.4.1.15. append_file

Syntax:
```
tError append_file(char *fname_in, char *fname_out);
```

This function appends the file (`fname_in`) to `fname_out`. If it is called several times it will always put the content of `fname_in` to the end of the file `fname_out`.

- ERR_CODE_ERROR_WRITING_FILE
- ERR_CODE_ERROR_READING_FILE
- ERR_CODE_NO_ERROR

4.4.1.16. copy_file

Syntax:
```
tError copy_file(char *fname_in, char* fname_out);
```

This function copies the file `fname_in` to the file `fname_out.`

- ERR_CODE_ERROR_WRITING_FILE
- ERR_CODE_ERROR_READING_FILE
- ERR_CODE_NO_ERROR

4.4.1.17. freeNestedData

Syntax:
```
void freeNestedData(void **objectID);
```

This function destroys the Object that was used to read a Nested Dataset without releasing the memory structure allocated and returned with the execution of `readNestedData()`.

4.4.2. Retrieving Functions

The following functions require that the `load_data_file()` function have already been called in order to obtain the `object_ID` associated with object of BinX, that contains the required variable.

An example of how can these functions be used to retrieve information is shown in Figure 4.

4.4.2.1. get_bit

Syntax:
```
tError get_bit(int object_ID, char *varname, tStartBit StartBit, int bit_nr, char *value);
```

This function requests the value of the bit at position `bit_nr` of the variable identified by `varname` in the object of BinX, identified by `object_ID`. Then the value is converted to char ‘0’ or ‘1’ and is returned to the user in `value`.

The function returns a t`Error` structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.2. get_bits

Syntax:
```
tError get_bits(int object_ID, char *varname, int bit_start_nr, int bit_stop_nr, const char
**value);
```

This function requests the value of the bits at positions between `bit_start` and `bit_stop` of the variable identified by `varname` in the object of BinX, identified by `object_ID`. Then the value is converted to a string of ‘0’ and ‘1’ and is returned to the user in `value`.

The function returns a t`Error` structure to inform the status of the operation.

NOTE: This function allocates memory for returning the string version of the bits and the user must release it when it is not used anymore.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.3. get_byte

Syntax:
```
tError get_byte(int object_ID, char *varname, char *value);
```

This function requests the value of the variable identified by `varname` in the object of BinX, identified by `object_ID`. Then the value is converted to char and is returned to the user in `value`. It is expected that the variable contains a number and that number is lower than 256.

The function returns a t`Error` structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.4. get_char

Syntax:
```
tError get_char(int object_ID, char *varname, char *value);
```

This function requests the value of the variable identified by `varname` in the object of BinX, identified by `object_ID`. Then the value is converted to char and is returned to the user in `value`.

The function returns a t`Error` structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.5. get_str

Syntax:
```
tError get_str(int object_ID, char *varname, char **value);
```

This function requests the value of the variable identified by `varname` in the object of BinX, identified by `object_ID`. Then the value is returned to the user as an array of char null terminated (a sting) in `value`.

The function returns a t`Error` structure to inform the status of the operation.

NOTE: This function allocates memory for returning the string and the user must release it when it is not used anymore.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT

- ERR_CODE_NO_ERROR

4.4.2.6. get_short

Syntax:
```
tError get_short(int object_ID, char *varname, unsigned short *value);
```

This function requests the value of the variable identified by `varname` in the object of BinX, identified by `object_ID`. Then the value is converted to short and is returned to the user in `value`.

The function returns a t`Error` structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.7. get_int

Syntax:
```
tError get_int(int object_ID, char *varname, int *value);
```

This function requests the value of the variable identified by `varname` in the object of BinX, identified by `object_ID`. Then the value is converted to int and is returned to the user in `value`.

The function returns a t`Error` structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.8. get_long

Syntax:
```
tError get_long(int object_ID, char *varname, long long *value);
```

This function requests the value of the variable identified by `varname` in the object of BinX, identified by `object_ID`. Then the value is converted to `long long` (C99 Standard) and is returned to the user in `value`.

The function returns a t`Error` structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.9. get_float

Syntax:
```
tError get_float(int object_ID, char *varname, float *value);
```

This function requests the value of the variable identified by `varname` in the object of BinX, identified by `object_ID`. Then the value is converted to float and is returned to the user in `value`.

The function returns a tError structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.10. get_double

Syntax:

```
tError get_double(int object_ID, char *varname, double *value);
```

This function requests the value of the variable identified by varname in the object of BinX, identified by object_ID. Then the value is converted to double and is returned to the user in value.

The function returns a tError structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.11. get_complex

Syntax:

```
tError get_complex(int object_ID, char *varname, tComplex *value);
```

This function requests the value of the variable identified by varname in the object of BinX, identified by object_ID. Then the value is converted to complex and is returned to the user in value.

The function returns a tError structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.12. get_doublecomplex

Syntax:

```
tError get_doublecomplex(int object_ID, char *varname, tDoubleComplex *value);
```

This function requests the value of the variable identified by varname in the object of BinX, identified by object_ID. Then the value is converted to double complex and is returned to the user in value.

The function returns a tError structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.13. get_struct (under development)

Syntax:

```
tError get_struct(int object_ID, char *varname, void *struct_var, int *struct_size);
```

This function returns a pointer to the structure associated to the variable identified in the schema by `varname`. This structure is defined on the schemas, but **must** be known by the library when invoked.

The function returns a t`Error` structure to inform the status of the operation.

SPECIAL NOTE: As opposed to the previous access methods, the structure accessed through this one may not change in runtime (i.e. altering the BinX schema), as the structure NEEDS to be known beforehand to the function invoking the CFI. Any change in the structure format will ALWAYS require recompilation to avoid errors, not in the CFI, but in the functions using it.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.14. get_fixed_array

Syntax:

```
tError get_fixed_array(int object_ID, char *varname, tVector *var_vector);
```

This function fills a structure of type `tVector` whose element `count` states the number of elements in the array pointed by the pointer in `tVector` element `value`. The memory pointed by `value` contains the values read from the file according to the schema. It is the responsibility of the caller to cast the `void*` in `value` to the correct structure, and **any change** in the **schema** related with the **fixed array must** be taken into account on the **caller's structure**.

The function returns a t`Error` structure to inform the status of the operation.

SPECIAL NOTE: As opposed to the previous access methods, the structure accessed through this one may not change in runtime (i.e. altering the BinX schema), as the structure NEEDS to be known beforehand to the function invoking the CFI. Any change in the structure format will ALWAYS require recompilation to avoid errors, not in the CFI, but in the functions using it.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR
- ERR_CODE_INVALID_DATA_TYPE
- ERR_CODE_ERROR_ALLOCATING_MEMORY

4.4.2.15. get_variable_array

Syntax:

```
tError get_variable_array(int object_ID, char *varname, tVector *var_vector);
```

This function fills a structure of type `tVector` whose element `count` states the number of elements in the array pointed by the pointer in `tVector` element `value`. The memory pointed by `value` contains the values read from the file according to the schema. It is the responsibility of the caller to cast the `void*` in `value` to the correct structure, and **any change** in the **schema** related with the **variable array must** be taken into account on the **caller's structure**.

The function returns a t`Error` structure to inform the status of the operation.

SPECIAL NOTE: As opposed to the previous access methods, the structure accessed through this one may not change in runtime (i.e. altering the BinX schema), as the structure NEEDS to be known beforehand to the function invoking the CFI. Any change in the structure format will ALWAYS require recompilation to avoid errors, not in the CFI, but in the functions using it.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR
- ERR_CODE_INVALID_DATA_TYPE
- ERR_CODE_ERROR_ALLOCATING_MEMORY

4.4.2.16. get_vector_byte

Syntax:

```
tError get_vector_byte(int object_ID, char* varname, tVectorChar** array_char);
```

This function is used to retrieve an array of `char`, and the number of elements in that array, contained in the variable with name `varname` associated with the object identified by `object_ID`.

The function returns a t`Error` structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.17. get_vector_short

Syntax:

```
tError get_vector_short(int object_ID, char* varname, tVectorShort* array_short );
```

This function is used to retrieve an array of `short`, and the number of elements in that array, contained in the variable with name `varname` associated with the object identified by `object_ID`.

The function returns a t`Error` structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.18. get_vector_int

Syntax:

```
tError get_vector_int(int object_ID, char* varname, tVectorInt* array_int );
```

This function is used to retrieve an array of `int`, and the number of elements in that array, contained in the variable with name `varname` associated with the object identified by `object_ID`.

The function returns a t`Error` structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.19. get_vector_long

Syntax:
```
tError get_vector_long(int object_ID, char* varname, tVectorLong* v_long);
```

This function is used to retrieve an array of `long long` (C99 Standard), and the number of elements in that array, contained in the variable with name `varname` associated with the object identified by `object_ID`.

The function returns a t`Error` structure to inform the status of the operation.

NOTE: The usage of the C99 Standard is required because a `long` in 32-bit architecture is 32-bit and it is expected to have a representation of 64-bit as it happens with the `long long` type provided by the C99 Standard

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.20. get_vector_float

Syntax:
```
tError get_vector_float(int object_ID, char* varname, tVectorFloat* array_float );
```

This function is used to retrieve an array of `float`, and the number of elements in that array, contained in the variable with name `varname` associated with the object identified by `object_ID`.

The function returns a t`Error` structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.21. get_vector_double

Syntax:
```
tError get_vector_double(int object_ID, char* varname, tVectorDouble* array_double);
```

This function is used to retrieve an array of `double`, and the number of elements in that array, contained in the variable with name `varname` associated with the object identified by `object_ID`.

The function returns a t`Error` structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.22. get_vector_doublecomplex

Syntax:
```
tError get_vector_doublecomplex(int object_ID, char* varname, tVectorDoublecomplex* array_dcomplex
);
```

This function is used to retrieve an array of tDoubleComplex, and the number of elements in that array, contained in the variable with name varname associated with the object identified by object_ID.

The function returns a tError structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.23. get_matrix_int

Syntax:
```
tError get_matrix_int(int object_ID, char* varname, tMatrixInt* matrix_int );
```

This function is used to retrieve a two-dimension array of int, and the number of elements in that array, contained in the variable with name varname associated with the object identified by object_ID.

The function returns a tError structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.24. get_matrix_byte

Syntax:
```
tError get_matrix_byte(int object_ID, char* varname, tMatrixByte* matrix_byte );
```

This function is used to retrieve a two-dimension array of byte values, and the number of elements in that array, contained in the variable with name varname associated with the object identified by object_ID.

The function returns a tError structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.25. get_matrix_float

Syntax:
```
tError get_matrix_float(int object_ID, char* varname, tMatrixFloat* matrix_float );
```

This function is used to retrieve a two-dimension array of `float`, and the number of elements in that array, contained in the variable with name `varname` associated with the object identified by `object_ID`.

The function returns a t`Error` structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.26. get_matrix_long

Syntax:
```
tError get_matrix_long(int object_ID, char* varname, tMatrixLong* matrix_long );
```

This function is used to retrieve a two-dimension array of `long`, and the number of elements in that array, contained in the variable with name `varname` associated with the object identified by `object_ID`.

The function returns a t`Error` structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.27. get_matrix_double

Syntax:
```
tError get_matrix_double(int object_ID, char* varname, tMatrixDouble* matrix_double );
```

This function is used to retrieve a two-dimension array of `double`, and the number of elements in that array, contained in the variable with name `varname` associated with the object identified by `object_ID`.

The function returns a t`Error` structure to inform the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.2.28. readNestedData

Syntax:
```
void* readNestedData(char *xml_fname, void **objectID);
```

This function is used to read the BinX file and create a memory structure that contains all the information retrieved from each element of the BinX file, particularly for files that are represented as linked-list in memory.

It receives an `xml_fname` that corresponds to the BinX file name, and an `objectID` where it is stored the reference to the Object that was used to read the BinX schema.

During its execution, this function allocates all the memory that is necessary to represent the information retrieved from the BinX file and returns a pointer to that memory structure.

For more details see Annex 5.

4.4.2.29. getDataFormat

Syntax:
```
char* getDataFormat(void *objectID);
```

This function is used to retrieve the string that describes the memory structure using a Pseudo Language, based on the BinX schema that describes the binary file. It receives a `void* objectID` that corresponds to the Object that was used to read the BinX schema, which means that this function ALWAYS require an previous invocation of `readNestedData()` to get the `objectID`.

For more details see Annex 5.

4.4.3. Storing Functions

An example of how can these functions be used to save information is shown in Figure 5.

4.4.3.1. save_data_file

Syntax:
```
tError save_data_file(int objectID, char *bin_filename);
```

This function saves to a binary file named `bin_filename` the information contained in an object of BinX, identified by `objectID`.

It Returns one of the enumerated `tError` values according to the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.2. set_bit (for future implementation)

Syntax:
```
tError set_bit(int object_ID, char *varname, tStartBit StartBit, int bit_nr, char value);
```

This function is used to set a bit at position `bit_nr` of value in variable `varname`, according to `value` that can be '0' or '1'.

StartBit defines the start of counting for `bit_nr` as described previously in section 4.2.3.

The function returns one of the enumerated `tError` values according to the status of the operation.

4.4.3.3. set_char

Syntax:
```
tError set_char(int object_ID, char *varname, char value);
```

This function receives a char (`value`) to be put on the variable `varname` of the BinX object identified by `objectID`. The `value` is converted to the format defined in the schema.

It returns one of the enumerated `tError` values according to the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.4. set_str

Syntax:
```
tError set_str(int object_ID, char *varname, char* value);
```

This function receives a null terminated string (`value`) to be put on the variable `varname` of the BinX object identified by `objectID`. The `value` is converted to the format defined in the schema.

It returns one of the enumerated `tError` values according to the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.5. set_short

Syntax:
```
tError set_short(int object_ID, char *varname, short value);
```

This function receives a short (`value`) to be put on the variable `varname` of the BinX object identified by `objectID`. The `value` is converted to the format defined in the schema.

It returns one of the enumerated `tError` values according to the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.6. set_int

Syntax:
```
tError set_int(int object_ID, char *varname, int value);
```

This function receives an int (`value`) to be put on the variable `varname` of the BinX object identified by `objectID`. The `value` is converted to the format defined in the schema.

It returns one of the enumerated `tError` values according to the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.7. set_long

Syntax:
```
tError set_long(int object_ID, char *varname, long long value);
```

This function receives a long (`value`) to be put on the variable `varname` of the BinX object identified by `objectID`. The `value` is converted to the format defined in the schema.

It returns one of the enumerated `tError` values according to the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.8. set_float

Syntax:

```
tError set_float(int object_ID, char *varname, float value);
```

This function receives a float (`value`) to be put on the variable `varname` of the BinX object identified by `objectID`. The `value` is converted to the format defined in the schema.

It returns one of the enumerated `tError` values according to the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.9. set_double

Syntax:

```
tError set_double(int object_ID, char *varname, double value);
```

This function receives a double (`value`) to be put on the variable `varname` of the BinX object identified by `objectID`. The `value` is converted to the format defined in the schema.

It returns one of the enumerated `tError` values according to the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.10. set_complex

Syntax:

```
tError set_complex(int object_ID, char *varname, tComplex value);
```

This function receives a complex (`value`) to be put on the variable `varname` of the BinX object identified by `objectID`. The `value` is converted to the format defined in the schema.

It returns one of the enumerated `tError` values according to the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.11. set_doublecomplex

Syntax:
```
tError set_doublecomplex(int object_ID, char *varname, tDoubleComplex value);
```

This function receives a double complex (`value`) to be put on the variable `varname` of the BinX object identified by `objectID`. The `value` is converted to the format defined in the schema.

It returns one of the enumerated `tError` values according to the status of the operation.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.12. set_vector_byte

Syntax:
```
tError set_vector_byte(int object_ID, char *varname, tVectorChar v_char);
```

This function receives an identification of an object already loaded (`object_ID`) and set the variable `varname` with the content of the char vector `v_char.` The type of this vector is `tVectorChar` and his definition is described in the section 3.3.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.13. set_vector_short

Syntax:
```
tError set_vector_int(int object_ID, char *varname, tVectorShort v_short);
```

This function receives an identification of an object already loaded (`object_ID`) and set the variable `varname` with the content of the short vector `v_short.` The type of this vector is `tVectorShort` and his definition is described in the section 3.3.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.14. set_vector_int

Syntax:
```
tError set_vector_int(int object_ID, char *varname, tVectorInt v_int);
```

This function receives an identification of an object already loaded (`object_ID`) and set the variable `varname` with the content of the int vector `v_int.` The type of this vector is `tVectorInt` and his definition is described in the section 3.3.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.15. set_vector_long

Syntax:
```
tError set_vector_long(int object_ID, char *varname, tVectorLong v_long);
```

This function receives an identification of an object already loaded (`object_ID`) and set the variable `varname` with the content of the long vector `v_long.` The type of this vector is `tVectorLong` and his definition is described in the section 3.3.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.16. set_vector_float

Syntax:
```
tError set_vector_float(int object_ID, char *varname, tVectorFloat v_float);
```

This function receives an identification of an object already loaded (`object_ID`) and set the variable `varname` with the content of the float vector `v_float.` The type of this vector is `tVectorFloat` and his definition is described in the section 3.3.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.17. set_vector_double

Syntax:
```
tError set_vector_double(int object_ID, char *varname, tVectorDouble v_double);
```

This function receives an identification of an object already loaded (`object_ID`) and set the variable `varname` with the content of the double vector `v_double.` The type of this vector is `tVectorDouble` and his definition is described in the section 3.3.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.18. set_vector_complex

Syntax:
```
tError set_vector_complex(int object_ID, char *varname, tVectorComplex v_complex);
```

This function receives an identification of an object already loaded (`object_ID`) and set the variable `varname` with the content of the complex vector `v_complex.` The type of this vector is `tVectorComplex` and his definition is described in the section 3.3.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.19. set_vector_doublecomplex

Syntax:
```
tError set_vector_doublecomplex(int object_ID, char *varname, tVectorDoublecomplex v_doublecomplex);
```

This function receives an identification of an object already loaded (`object_ID`) and set the variable `varname` with the content of the double complex vector `v_doublecomplex.` The type of this vector is `tVectorDoublecomplex` and his definition is described in the section 3.3.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.20. set_matrix_int

Syntax:
```
tError set_matrix_int(int object_ID, char *varname, tMatrixInt matrix_int);
```

This function receives an identification of an object already loaded (`object_ID`) and set the variable `varname` with the content of the `int` two-dimension array `matrix_int.` The type of this vector is `tMatrixInt` and his definition is described in the section 3.3.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.21. set_matrix_float

Syntax:
```
tError set_matrix_float(int object_ID, char *varname, tMatrixFloat matrix_float);
```

This function receives an identification of an object already loaded (`object_ID`) and set the variable `varname` with the content of the `float` two-dimension array `matrix_float.` The type of this vector is `tMatrixFloat` and his definition is described in the section 3.3.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.22. set_matrix_long

Syntax:
```
tError set_matrix_long(int object_ID, char *varname, tMatrixLong matrix_long);
```

This function receives an identification of an object already loaded (`object_ID`) and set the variable `varname` with the content of the `long` two-dimension array `matrix_long`. The type of this vector is `tMatrixLong` and his definition is described in the section 3.3.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.23. set_matrix_double

Syntax:

```
tError set_matrix_double(int object_ID, char *varname, tMatrixDouble matrix_double);
```

This function receives an identification of an object already loaded (`object_ID`) and set the variable `varname` with the content of the `double` two-dimension array `matrix_double`. The type of this vector is `tMatrixDouble` and his definition is described in the section 3.3.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.24. set_matrix_byte

Syntax:

```
tError set_matrix_byte(int object_ID, char *varname, tMatrixByte matrix_byte);
```

This function receives an identification of an object already loaded (`object_ID`) and set the variable `varname` with the content of the `byte` value two-dimension array `matrix_byte`. The type of this vector is `tMatrixByte` and his definition is described in the section 3.3.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.25. set_matrix_doublecomplex

Syntax:

```
tError set_matrix_int(int object_ID, char *varname, tMatrixDoublecomplex matrix_dcomplex);
```

This function receives an identification of an object already loaded (`object_ID`) and set the variable `varname` with the content of the `tDoubleComplex` two-dimension array `matrix_dcomplex`. The type of this vector is `tMatrixDoublecomplex` and his definition is described in the section 3.3.

The errors returned are:

- ERR_CODE_ERROR_ACCESSING_OBJECT
- ERR_CODE_NO_ERROR

4.4.3.26. writeNestedData

Syntax:

```
void writeNestedData(void *data, char *xml_fname, char *fname);
```

This function is used to write to a binary file `fname` the memory structure `data`, described in the BinX schema `xml_fname`. Its main intent is to write structures with nested variable arrays

For more details see Annex 5.

5. ANNEX 1 – DISK DATA ACCESS

In order to select the best way to access data from a hybrid XML file, a performance test was created by defining a binary file which contains 4000 scenes and a program was written to perform two different methods of accessing data:

In method A an XML-schema is created containing the 4000 schemas of the scenes and the binary file is read into memory using this huge XML-schema, which means that the binary file is completely loaded into memory. Then each one of the N scenes is accessed, reading a value of that scene.

In method B a chunk of data, corresponding to one of the N scenes, is extracted into another binary file and afterward loaded into memory using the XML-schema of the scene. Then a value of that scene is read from memory.

In this test, each scene was comprised of a total of 11 Kbytes, and the test results are shown in Table 5.

Nr of Kbytes (N)	Method A (sec)	Method B (sec)
11	1.7	0.015
1100	1.72	0.325
5500	1.72	1.7
11000	1.74	3.2
22000	2.21	6.36
40000	5	12.9

Table 5: Time used accessing a variable of N scenes (in seconds).

The machine used to perform the tests has the following characteristics:

CPU: Intel Pentium 4 2,4 GHz

RAM: 470 MB DDRAM 333MHz

Disk: IDE 36 GB – UDMA 100 I/O – 50,39 MB/s

From this analysis, the conclusion obtained was that there should be two ways to prepare the binary file and schema for read access, depending on the size of the file handled. The BinXML-FH provides a default “auto” mode which allows it to choose which method is best suited for the task requested, and it also provides a “manual” mode which allows the user to choose which method to use. All this is still under development.

6. ANNEX 2 – BINX OBJECT DESCRIPTION

This library use BinX API to handle binary data files, by defining XMLSchemas describing the content of these files.

In this section it is described the objects and methods of BinX that are used.

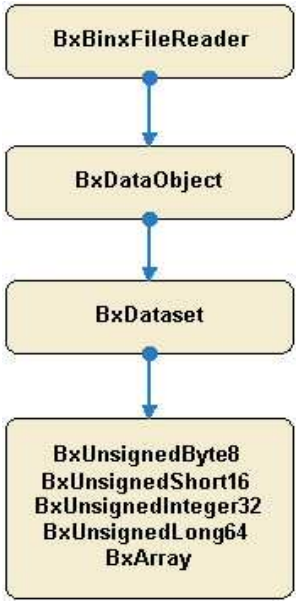

Figure 6: BinX data structure.

6.1. Used Types

- **BxBinxFileReader**: Read and parse a BinxXML file.
- **BxDataset**: Ordered collection of objects of several BinX data types.
- **BxDataObject**: Class that implements all BinX data types.
- **BxUnsignedByte8**: BinX data type 8-bit unsigned byte.
- **BxUnsignedShort16**: BinX data type 16-bit unsigned integer.
- **BxUnsignedInteger32**: BinX data type 32-bit unsigned integer.
- **BxUnsignedLong64**: BinX data type 64-bit unsigned long.
- **BxArray**: Base class for all BinX array types.

6.2. Used Methods

- **BxArray::get(int index)** - Get a copy of a single data element from an array.

- **BxDataset::getArray(int index)** - Get a reference to the BxArray instance in the current position of the ordered list of dataset members.
- **BxDataset::getDataset()** - Get a reference to the BxArray instance in the current position of the ordered list of dataset members.
- **BxDataset::getDataObject(char* Name)**
- **BxDataset::getDataObject(int index)** - Get a reference to an object in a dataset, based on the user-defined variable name of that object or its relative position in the dataset.
- **BxDataset::getUnsignedShort16(char* varName);**
- **BxDataset::getUnsignedShort16(int index) -** Get a reference to a BxUnsignedShort16 object contained in a dataset, based on its variable name or relative position.
- **BxDataset::getUnsignedLong64(char* varName);**
- **BxDataset::getUnsignedLong64(int index)** - Get a reference to a BxUnsignedLong64 object contained in a dataset, based on its variable name or relative position.
- **BxDataset::addDataObject()** - Append a specified instance of BxDataObject (like BxArray, BxUnsignedByte8 ...) to a dataset.
- **BxDataset::readFromFile()** - Read the application data values for all dataset member objects from the associated binary file.
- **BxDataset::toStreamBinary(File* filename, BxByteOrder)** - Write the application data value of each of the dataset member objects to a specified binary file with the specified byte order.

7. ANNEX 3 – EARTH EXPLORER FILE HANDLING CFI

The BinXML-FH library uses the EECFI to access the ASCII XML information.

In this section it is described the functions that are used.

7.1. Methods Used

- **void xf_tree_init_parser(char* file, long* error)** - Loads an XML file into memory.
- **void xf_tree_read_string_element_value(long *fd, char *element, char *value, long *error)** - Reads a string value.
- **void xf_tree_read_string_attribute(long *fd, char *element, char *attribute_name, char *attribute_value, long *error)** - Reads an attribute as string.
- **void xf_tree_add_child(long *id, char *parent, char *name, long *error)** - Add a new element to an XML document as a child of parent.
- **void xf_tree_add_attribute(long *id, char *current, char *name, long *error)** - Add a new attribute carried by an element.
- **void xf_tree_set_string_node_value(long *id, char *name, char *value, char *format, long *error)** - Set a string node value.
- **void xf_tree_write(long *id, char *name, long *error)** - Write the data to a file on disk.
- **void xf_tree_cleanup_parser (long *fd, long *error)** - Releases the memory resources taken by the XML parser for all the open files.
- **void xf_tree_cleanup_all_parser()** - Gets the name of the current element.

8. ANNEX 4 – SELF TEST CASES

8.1. test_binxml

The BinXML-FH library provides a test program to check if it is everything correct with the installation of the library itself and with the dependent libraries.

The test program is located in “test/bin/test_binxml” and the source code in “test/src/test_binxml.c”. The main goal of this program is to use all the set and get functions available, it produces a hybrid file “data_verify.eef” (58009 bytes) in “test/data” directory.

For testing the reading of variable arrays it uses another hybrid file “var_array.EEF” (757 bytes) located in the same directory.

The values used are the following:

Short	300
Int	3000
Float	350.543
Long	3500768
Double	3500.10
Character	d
Byte	100
String	Test binxml-fh
Complex	R=0.2 I=0.6
Doublecomplex	R=0.258 I=0.636
Short_vector	567
Int_vector	2345
Byte_vector	86
Long_vector	6745662
Float_vector	54637.8
Double_vector	0.155555
Doublecomplex_vector	R=0.155555 I=0.255555
Complex_vector	R=0.3 I=0.4
Int_matrix	6454233
Float_matrix	33244.5
Long_matrix	28876544
Double_matrix	0.3343

Byte_matrix	35
Variable array of basic type	6 integers (2,4,8,16,32,64)
Variable array of structure type	2 structures of 3 integer (2,4,8) and (32,64,128)

The self-test shall output the results on screen, producing a report like the following one:

```
Reading Config File...OK!
Getting the variable_array of basic type value:  ...OK
 -----------------------------------------------------
Getting the variable_array of structure type value:  ...OK
 -----------------------------------------------------
Creating object for write  ....OK
Object size is: 55166  ....OK

 Setting object types .....

set_short: 300 ....OK
 -----------------------------------------------------
set_int: 3000 ....OK
 -----------------------------------------------------
set_float: 350.543 ....OK
 -----------------------------------------------------
set_long: 3500768 ....OK
 -----------------------------------------------------
set_double: 3500.10 ....OK
 -----------------------------------------------------
set_character: d ....OK
 -----------------------------------------------------
setting_byte_vector: 86 ....OK
 -----------------------------------------------------
setting_short_vector: 567 ....OK
 -----------------------------------------------------
setting_int_vector: 2345 ....OK
 -----------------------------------------------------
set_byte: 100 ....OK
 -----------------------------------------------------
set_string: test binxml-fh ....OK
 -----------------------------------------------------
set_complex: r=0.2 i=0.6 ....OK
 -----------------------------------------------------
set_doublecomplex: r=0.258 i=0.636 ....OK
 -----------------------------------------------------
setting_long_vector: 6745662 ....OK
 -----------------------------------------------------
setting_float_vector: 54637.8 ....OK
 -----------------------------------------------------
setting_double_vector: 0.1555555 ....OK
 -----------------------------------------------------
setting_doublecomplex_vector: r=0.1555555 i=0.2555555 ....OK
 -----------------------------------------------------
setting_complex_vector: r=0.3 i=0.4 ....OK
 -----------------------------------------------------
setting_int_matrix: 6454233 ....OK
 -----------------------------------------------------
setting_float_matrix: 33244.5 ....OK
 -----------------------------------------------------
setting_long_matrix: 28876544 ....OK
 -----------------------------------------------------
setting_double_matrix: 0.3343 ....OK
 -----------------------------------------------------
setting_byte_matrix: 35 ....OK
 -----------------------------------------------------
Testing save_data_file: ...OK
```

```
 ------------------------------------------------------
Testing free_data_file: ...OK
 ------------------------------------------------------

Producing hybrid file: ...OK
 ------------------------------------------------------


**********************
Reading Written Data
**********************
Extracting Header: ...OK
 ------------------------------------------------------

Getting Data File Info: ...OK
 ------------------------------------------------------

Extracting Data: ...OK
 ------------------------------------------------------

Preparing Schema: ...OK
 ------------------------------------------------------

Loading Data File: ...OK
 ------------------------------------------------------

Getting the Short Value: 300 ...OK
 ------------------------------------------------------
Getting the Int Value: 3000 ...OK
 ------------------------------------------------------
Getting the Float Value: 350.543 ...OK
 ------------------------------------------------------
Getting the Long Value: 3500768 ...OK
 ------------------------------------------------------
Getting the Double Value: 3.500100e+03 ...OK
 ------------------------------------------------------
Getting the Character Value: d ...OK
 ------------------------------------------------------
Getting the Byte Value: 80 ...OK
 ------------------------------------------------------
Getting the String Value: test binxml-fh ...OK
 ------------------------------------------------------
Getting the Complex Value: r=0.200000 i=0.600000 ...OK
 ------------------------------------------------------
Getting the DoubleComplex Value: r=0.258000 i=0.636000 ...OK
 ------------------------------------------------------
Getting the vector_char Value: ...OK
 ------------------------------------------------------
Getting the vector_short Value: ...OK
 ------------------------------------------------------
Getting the vector_int Value: ...OK
 ------------------------------------------------------
Getting the vector_long Value: ...OK
 ------------------------------------------------------
Getting the vector_float Value: ...OK
 ------------------------------------------------------
Getting the vector_double Value: ...OK
 ------------------------------------------------------
Getting the vector_doublecomplex Value:  ...OK
 ------------------------------------------------------
Getting the vector_complex Value:  ...OK
 ------------------------------------------------------
Getting the matrix_int Value: ...OK
 ------------------------------------------------------
Getting the matrix_float Value: ...OK
 ------------------------------------------------------
Getting the matrix_long Value: ...OK
 ------------------------------------------------------
Getting the matrix_double Value: ...OK
```

```
 ------------------------------------------------------
Getting the matrix_byte Value: ...OK
 ------------------------------------------------------
Getting the Bit 5 of Short Value: 1 ...OK
 ------------------------------------------------------
Getting the Bits 0 to 5 of Short Value: 101100 ...OK
 ------------------------------------------------------
Testing free_data_file: ...OK
 ------------------------------------------------------
```

8.2. test_nested

The library functionality with nested complex data types – e.g. arrays (fixed or variable) that include other arrays – is tested with “test_nested”, while “test_binxml” tests the functionality with primitive BinX data types and with arrays of primitive types.

The test program binary is “test/bin/test_nested”, which corresponds to the “test/src/test_nested.c” source file. The main purpose of this test is verifying that nested data structures of different complexity and endianness can be read from and written to disk. If both read and write operations complete without errors, the files are compared. If they have no differences, the test is considered successful.

The program uses XML and data files (DBL) nested5_b, nested5_l, nested6_b, nested6_l and nested7, all of them located in the “test/data” directory; nested5_b.dbl and nested6_b.dbl have data in big-endian byte ordering, while the others have it in little-endian byte ordering. A nestedout.dbl is created in the “data” directory during the execution.

If we consider the BinX data to be arranged in a tree-like structure, with complex data types as arrays at higher levels and primitive data types at lower levels, the data structures used by test_nested have arrays at three levels. The important differences between them are the array types (fixed or variable) at each level. The data structure of nested5_b and nested5_l files has a variable array at the highest level, another variable array at the second level and a fixed array at the third level. The data structure of nested6_b and nested6_l files differs because it has a fixed array at the highest level. The data structure of nested7 has variable arrays at all the three levels.

The test should output to the screen a report like the following:

```
Data File:      unused.dbl
Schema File:    test/data/nested5_b.xml
Size of dataset = 61
Size of buffer with pointers = 9
Testing test/data/nested5_b.xml        format: "#BP{B#SP(2(BI)P)SP}"          OK
Data File:      unused.dbl
Schema File:    test/data/nested5_l.xml
Size of dataset = 61
Size of buffer with pointers = 9
Testing test/data/nested5_l.xml        format: "#BP{B#SP(2(BI)P)SP}"          OK
Data File:      unused.dbl
Schema File:    test/data/nested6_b.xml
Size of dataset = 61
Size of buffer with pointers = 27
Testing test/data/nested6_b.xml        format: "B2{B#SP(2(BI)P)S}"            OK
Data File:      unused.dbl
Schema File:    test/data/nested6_l.xml
Size of dataset = 61
Size of buffer with pointers = 27
```

```
Testing test/data/nested6_l.xml        format: "B2{B#SP(2(BI)P)S}"             OK
Data File:      unused.dbl
Schema File:    test/data/nested7.xml
Size of dataset = 426
Size of buffer with pointers = 12
Testing test/data/nested7.xml          format: "#IP{SIII#SP{FFFFF#SP(SP)P}P}"  OK
```

8.3. file_transform

Another program provided is “bin/file_transform” and the purpose of this is to split hybrid files into header and data block or join them into a hybrid file.

To split a hybrid file the test must run with the following arguments:

- test/bin/file_transform ../binxml-config.xml split data/data_verify.eef

To join a header file with a datablock the command is the following:

- test/bin/file_transform ../binxml-config.xml join data/data_verify.eef

9. ANNEX 5 – NESTED VARIABLE ARRAYS

The support for nested variable arrays is a new functionality that is available in release 3.4 of BinXML File Handing.

9.1. Purpose

Since the introduction of variable arrays support in this tool it has been studied the possibility of migrating the schemas definition to contain exclusively variable arrays, which leads to the implementation of "nested" variable arrays.

By definition, *a **variable array** is a structure that uses information read from the binary file to define the actual size of the array.*

Introducing the variable arrays on the dataset definition means that part of the information relevant to the Dataset Size will be inside the Dataset itself.

Let's see an example with the proposed changes to the schemas.

Considering the following schema in the current format:

```
<binx>

    <definitions>
        <defineType typeName="Map_Type">
            <struct>
                <arrayFixed varName="layer">
                    <float-32 varName="value"/>
                    <dim indexTo="720">
                        <dim indexTo="1440"/>
                    </dim>
                </arrayFixed>
            </struct>
        </defineType>
    </definitions>

    <dataset src=" " byteOrder="bigEndian">
        <useType typeName="Map_Type" varName="Galaxy_Map[#]"/>
    </dataset>

    <templateSize unit="bytes">4155844</templateSize>
</binx>
```

Types definition

Datasets definition

The previous schema could be adapted to follow the format using variable arrays by:

1) Using the same type definition and changing the dataset definition.

```
<binx>
    <definitions>
        <defineType typeName="Galaxy_Map_Type">
            <struct>
                <arrayFixed varName="layer">
                    <float-32 varName="value"/>
                    <dim indexTo="720">
                        <dim indexTo="1440"/>
                    </dim>
                </arrayFixed>
            </struct>
        </defineType>
    </definitions>
    <dataset src=" " byteOrder="bigEndian">
        <arrayVariable varName="Galaxy_Map">
            <sizeRef>
                <unsignedByte-8/>
            </sizeRef>
            <useType typeName="Galaxy_Map_Type"/>
            <dim/>
        </arrayVariable>
    </dataset>
</binx>
```

Types definition

Datasets definition

2) Using the same dataset definition and creating a new type definition.

```
<binx>
    <definitions>
        <defineType typeName="Galaxy_Map_Type">
            <struct>
                <arrayFixed varName="layer">
                    <float-32 varName="value"/>
                    <dim indexTo="720">
                        <dim indexTo="1440"/>
                    </dim>
                </arrayFixed>
            </struct>
        </defineType>
        <defineType typeName="Galaxy_Map_Record">
            <struct>
                <arrayVariable varName="record">
                    <sizeRef>
                        <unsignedByte-8/>
                    </sizeRef>
                    <useType typeName="Galaxy_Map_Type"/>
                    <dim/>
                </arrayVariable>
            </struct>
        </defineType>
    </definitions>
    <dataset src=" " byteOrder="bigEndian">
        <useType typeName="Galaxy_Map_Record" varName="Galaxy_Map"/>
    </dataset>

</binx>
```

Types definition

Dataset definition

Another objective related with the proposed changes to the schemas would be avoiding the obligation of replicating the data element in the schema before reading data, i.e., instead of calling the `prepare_schema()` function [see 4.4.1.10], the function `prepare_schema_nested()` would be called [see 4.4.1.11], to insert only the proper data set source and byte order attributes.

With this change the diagram of Figure 3 would be simplified as shown in Figure 7.

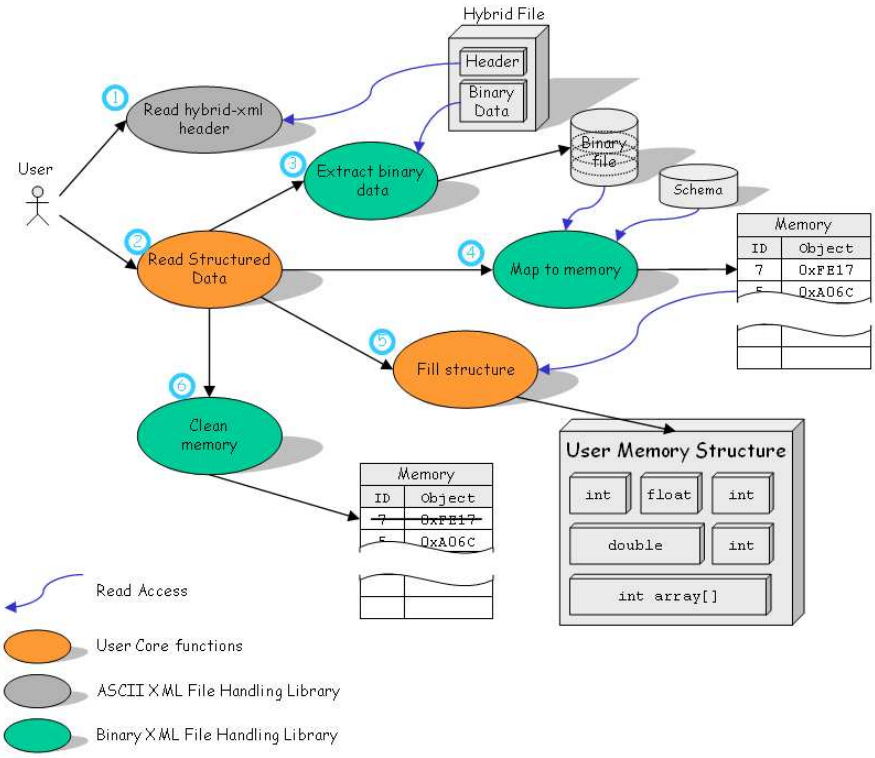

Figure 7: Retrieving data from a hybrid XML data file using the variable array approach.

The procedure of reading data is almost the same, with a minor change that is the introduction of a counter (number of elements in the variable array) at the beginning of the extracted binary file as shown in Figure 8.

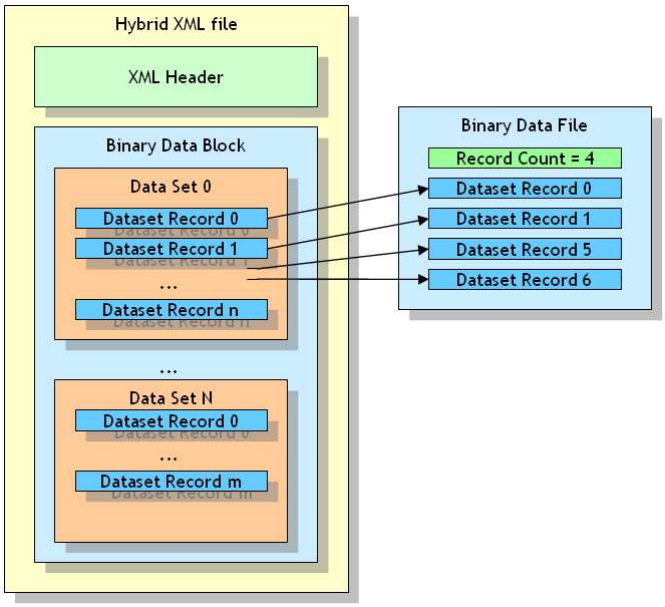

Figure 8: Generating the binary file using variable array approach.

9.2. Limitations

There have been identified a couple of limitations related to the usage of nested variable arrays:

1. Format:
   1.1. Variation of structures: changes in the order by which the elements of the structure are defined in memory must be followed with the corresponding change in the order by which the elements are defined in the schema, and *vice versa*. The same is applicable to any change in the type of data, either in memory structure or schema file, which implies a recompilation of user's code.
2. Writing data:
   2.1. Writing the data is done always according to the specified format. Any mistake in the format description may cause a segmentation fault since that is the only way that BinXML-FH can know the format of the memory structure.
3. Reading data:
   3.1. The data read from the BinX file always follows the same approach to produce the memory structure, which means that the BinXML user's memory structure must be defined accordingly.

9.3. Approach

In C++ there is a way of knowing the class of a specific object, and in order to mimic this functionality in C it is necessary to create a workaround.

To inform BinXML about the "format" of the User's data structure it is necessary to use a *pseudo-language* that can be used to validate the schema data types against the User's defined memory structure.

The pseudo-language has the following element:

Element	Description
B, **S**, **I**, **L**, **F** and **D**	Represent the basic types byte, short, int, long, float and double
0 – 9	Represent the number of elements of a BinX Fixed Array and correspond to a static allocation in the memory structure, i.e., `array[4]` for example.
{, **(**, **)** and **}**	Are used to enclosure a repeating block, which is typically an array element. ( ) are used to enclosure a fixed size element structure and { } a variable size structure, i.e., a structure that contains variable sized arrays (or linked-list).
P	Represent a pointer.
#	Is associated with variable arrays and represent a number that must be read from the buffer. When # is found on the *format* string it appears always in the sequence **#?P** where **?** can be one of **B**, **S**, **I** or **L**, and represent the *type* of the number that is used to represent the array counter.

Rules:

A Variable array is always translated in a linked-list memory structure, and *vice-versa*, which means that always, after the description of the Variable Array Element structure, a new pointer must be inserted to point to the next element of the Variable Array.

Therefore, to describe a Variable Array of integers with a counter defined by a Byte, it should be used a *format* "#BP(IP)". If instead of an integer, the array element is a complex structure, let say a fixed array of 3 floats, the *format* should be "#BP(3(F)P)".

To describing a fixed array of 4 elements, each one being a variable array of Integers with a counter defined by a short, the *format* should be "4{#SP(IP)}".

Example: Considering the following user memory structure and the corresponding BinX schema:

```
typedef struct _data {
   float     valf;
   int       vali;
   double    vald;
   struct _data *next;
} __attribute__ ((packed)) t_data;

Typedef struct {
   int       count;
   t_data    *array;
} __attribute__ ((packed)) t_myarray;
```

```
<?xml version="1.0" encoding="UTF-8"?>
<binx>
   <definitions>
      <defineType typeName="Array_Type">
         <struct>
            <float-32 varName="f"/>
            <integer-32 varName="i"/>
            <double-64 varName="d"/>
         </struct>
      </defineType>
      <defineType typeName="Data_Type">
         <struct>
            <arrayVariable varName="record">
               <sizeRef>
                  < unsignedInteger-32/>
               </sizeRef>
               <useType typeName="Array_Type"/>
               <dim/>
            </arrayVariable>
         </struct>
      </defineType>
   </definitions>
   <dataset src=" " byteOrder="bigEndian">
      <useType typeName="Data_Type" varName="My_Array"/>
   </dataset>
</binx>
```

The string describing the *format* of the memory structure is "**#IP(FIDP)**"

What about nested variable arrays?

Example: Considering the following user memory structure and the corresponding BinX schema:

```
typedef struct {
    char polID;
    int polVal;
} __attribute__ ((packed)) tPolData;

typedef struct s_VarArrayFixedType {
    tPolData polFixArray[2];
    struct s_VarArrayFixedType *next;
}__attribute__ ((packed)) tVarArrayFixedType;

typedef struct s_dstype {
    char   charElement;
    short  varArrayFixedTypeCount;
    tVarArrayFixedType  *varArrayFixedType;
    short  shortElement;
    struct s_dstype  *next;
}  __attribute__ ((packed)) dstype_n5;

typedef struct {
    byte count;
    dstype_n5 *dataset;
}  __attribute__ ((packed)) tProduct_n5;
```

```
<?xml version="1.0" encoding="UTF-8"?>
<binx xmlns="http://www.edikt.org/binx/2003/06/binx">
  <definitions>
    <defineType typeName="tPolFixArray">
      <struct>
        <arrayFixed varName="PolFixArray" byteOrder="littleEndian">
          <struct>
            <byte-8 varName="polID" byteOrder="littleEndian" />
            <integer-32 varName="polVal" byteOrder="littleEndian" />
          </struct>
          <dim indexTo="1"></dim>
        </arrayFixed>
      </struct>
    </defineType>
    <defineType typeName="dstype">
      <struct>
        <byte-8 varName="charElement" byteOrder="littleEndian" />
        <arrayVariable varName="varArrayFixedType"
                       byteOrder="littleEndian">
          <sizeRef>
            <short-16 byteOrder="littleEndian" />
          </sizeRef>
          <useType typeName="tPolFixArray"
                   varName="typedFixArrayElement"
                   byteOrder="littleEndian" />
          <dim/>
        </arrayVariable>
        <short-16 varName="shortElement" byteOrder="littleEndian" />
      </struct>
    </defineType>
  </definitions>

  <dataset src="nested5_l.dbl" byteOrder="littleEndian">
    <arrayVariable varName="dsarray" byteOrder="littleEndian">
      <sizeRef>
        <byte-8 byteOrder="littleEndian" />
      </sizeRef>
      <useType typeName="dstype" varName="dsname"
        byteOrder="littleEndian" />
      <dim></dim>
    </arrayVariable>
  </dataset>
</binx>
```

The *format* describing this memory structure is "**#BP{B#SP(2(BI)P)SP}**"

Does it works for writing data?

The memory *format* is of utmost importance for writing data, since that is the only way to access the information, namely, reading the counters of variable arrays.