The Universal Access Layer User Guide

G. Manduchi, F.Imbeaux, M. Haefele

03/03/2009

1. Introduction

In the framework of the Integrated Tokamak Modelling (ITM) task, the Universal Access Layer (UAL) provides the capability of storing/retrieving data involved in simulation. The underlying data structure is hierarchical, thus allowing groups of related data items to be associated in subtrees of the main data tree. The granularity in data access is given by the definition of a set of Consistent Physical Objects (CPOs). Every CPO is represented by a subtree in the underlying data hierarchy. CPOs represent the basic data set which can be read/written. In other words, it is not possible to read/write data items associated with a given CPO in insulation, it is always necessary to access the whole set of data stored in the subtree associated with the CPO.

It is worth noting that physics programs do not need to access the UAL library when they are run under the control of the KEPLER simulation framework. In this case, in fact, data is passed as argument to the physics programs, which in turn return computed results. Nevertheless, the granularity introduced by CPOs is retained, that is physics programs will receive and return whole CPOs, not parts of them.

The rest of this document refers to the usage of the UAL outside the KEPLER framework and is organized as follows:

- ***General concepts*** introduces the architecture of the UAL and described those aspects which are common to all the binding languages. It is followed by language-specific interface description, in particular:
  - *Fortran*;
  - *C++*;
  - *Python*;
  - *Java*.

- ***Low level interface description*** is intended for developer of new language bindings or software tools, and describes the C interface to the common, low-level data access routines.

2. General Concepts

2.1 XML definition

The UAL provides an interface to data handleld in ITM simulation in a Multilanguage framework. For this reason, the definition of the structures forming the CPOs involved in simulation is expressed using language-independent representation, that is *XML Schema Description*.
XML Schema description is a World Wide Web Consortium (W3C) standard and is designed to describe a set of rules to which a class of XML descriptions has to adhere. In this context, the XML Schema description specifies the rules describing the structure and the allowed data types of the ITM database. Observe that, despite the fact that XML Schemas have been designed to specify the structure of XML descriptions, the ITM database will very likely not be defined as an XML file. However users are not concerned on the actual implementation of the database, rather they need to know how data are logically organized, and the XML Schema description provides an unambiguous

way of describing the logical data organization. The current definition of the database structure, i.e. which CPOs are defined and their structure is available in the ITM Web page.

In order to store/retrieve CPOs, it is necessary do define for each CPO the corresponding structure for the given programming language. Although the XML Schema description is not related to any programming language, it is possible to build automated translation tools which produce the language-specific CPO structure definition, as well as the Application Programming Interface (API) implementation in each language, starting from the XML Schema Description (XSD). The translation tools are based on the XSLT specification, a W3C standard designed to translate XML documents. The result of the translation is the language-specific interface which will be used in user programs.

2.2 Time-dependent and Time-independent CPO fields

In general a CPO can contain both time-dependent and time-independent information. Typically, information related to the tokamak hardware will not be time-dependent, while the value of plasma physical quantities will likely be time dependent. Therefore the fields of a CPO can be either time-dependent or time-independent. The CPO itself will be time-independent if it contains only time-independent fields. It will be time-dependent otherwise. Only few CPOs in the ITM database will be time-independent (e.g. *topinfo*, *summary, vessel*), while the others will describe physical phenomena which vary over time.
Time-dependent CPOs are treated as arrays of the elementary CPO structure, i.e. in Fortran : equilibrium(:) is a pointer and equilibrium(i) is an equilibrium structure corresponding to time index i. Since many physics code manipulate only one time slice at a time, special UAL functions exist to extract a single time slice of the CPO : these are the GET_SLICE and PUT_SLICE functions.

2.3 Common operations

ITM databases borrow the philosophy of the Pulse Files in Nuclear Fusion experiment. As pulse files typically contain all the information which is relevant to a plasma discharge, an ITM database will contain all the information which is related to the execution of a given simulation. A database model defines the hierarchical structure of the ITM simulation databases. The ***create()*** operation will create a new empty database taking the internal organization (i.e. the CPO structures) from the ITM database model. Two integer numbers define the newly created ITM simulation database: the *Shot* and the *Run* numbers. The Shot number is intended to refer to the actual shot number of the experiment on which the simulation is based. The Run number is an additional index to list all simulations referring to a given Shot number. Internally to the UAL, these two numbers are combined to form a Generalised Pulse Number, i.e. an extension of the usual shot number in Nuclear Fusion experiments. It means that the UAL functions ***create*** and ***open*** use these two numbers as input. Nevertheless, from the user point of view, a simulation is defined by additional information such as the Machine name (name of the experiment it refers to) and the User name (public or private user ITM database). These two parameters define a folder in the ITM Gateway in which all Shot and Run numbers for the given (Machine, User) are stored. To run the UAL properly, environment variables must have been set to the folder corresponding to Machine and User. This operation is done outside the UAL, which manages only the Shot and Run numbers inside a given (Machine, User) folder. A relational database and additional libraries will be used to set the environment variables properly prior calling the UAL. This is outside the scope of this document.

Besides creating new (empty) ITM simulation databases, it is also possible to open an existing simulation database, using the ***open()*** operation, again specifying the Shot and Run numbers.

The ***get()*** and ***put()*** functions are used to read and write a whole CPO. This is straightforward when the CPO does not depend on time. Conversely, when the CPO is time-dependent, all time slices are read (resp. written) to (resp.from) an array of elementary CPO structures. These CPO contain a ***time*** field, located at their root (i.e. for Fortran, cpo%time), which contains the time value to which all time dependent fields of the CPO refer. During a get() operation, time-independent fields of the time-dependent CPO are filled for every time value : cpo(i)%my_field is filled for all i values with the same time-independent value. It is worth noting that time-independent CPO fields are internally stored in a single instance in order to save space in the simulation database. The ***get()*** operation does all necessary memory allocations for all CPO fields and for the CPO array itself.
During a ***put()*** operation, time-independent fields of the time-dependent CPO are written using the value of the first time index of the CPO : cpo(1)%my_field is used as the time-independent value of this field.
Note that a ***put()*** operation erases any values of CPO fields that could have been written before.

The ***getSlice()*** and ***putSlice()*** functions are used to read and write a single time slice of a CPO.
In the ***getSlice()*** case, it is necessary to provide also the value of the time wanted. If the passed time value does not fit exactly the time stored in any CPO sample, the returned sample is chosen based on the passed interpolation mode. The following interpolation modes are supported in the UAL:

- 1 : *CLOSEST_SAMPLE*: the returned CPO is the stored CPO whose time is closest to the passed time;
- 2 : *PREVIOUS_SAMPLE*: the CPO whose time is just before the passed time is returned.
- 3 : *LINEAR INTERPOLATION*: the values of the time-dependent return CPO fields are computed according to a linear interpolation between the corresponding values of the CPOs just before and after the passed time;

If the passed time lies outside the time range of the stored CPOs, the first (last) CPO stored sample in the range is returned.

Using operation ***putSlice()*** allows appending a new time slice to an already stored time-dependent CPO. Note that the ***putSlice()*** function affects only the time-dependent fields of the CPO. Therefore, there are two possibilities :

- at least one time slice of the CPO is previously written by a ***put()*** operation. Then the time-independent fields are properly written and ***putSlice()*** can safely be used to append time slices to this CPO.
- The first time slice of the CPO is written using a ***putSlice()*** operation. In this case, the CPO time-independent field must be properly filled by using the ***putNonTimed()*** instruction. This one will write only the time-independent fields of the CPO and must be followed by a ***putSlice()*** operation.

It is also possible to replace the last CPO time slice stored in the simulation database, via operation ***replaceLastSlice().*** It is instead not possible to replace a CPO time slice other than the last one, without overwriting the whole CPO array.

2.4 CPO location within the dataset

A dataset of the ITM database is defined by the four elements (User, Machine, Shot, Run). A dataset consists in an instance of the whole ITM data structure, i.e. a collection of CPOs. For each CPO name (e.g. topinfo, equilibrium, …), a certain number of occurrences of this CPO can exist within a given dataset. A maximum number of occurrences is defined for each CPO in the XML

Schema description. Note that CPO occurrences are completely independent of the others : although they have the same structure, they contain a priori different data and may have different time bases.

UAL functions make use of the *XPath* standard to specify CPO paths and field paths inside a CPO (path names in XPath are quite similar to file names in Unix file system). XPath is a W3C standard designed to unambiguously specify items within a XML (hierarchical) description. All CPOs are defined at the root level, i.e. their name in the database hierarchy is simply their name. For example, the ***equilibrium*** CPO is defined by the simple path name *equilibrium*. Although CPO fields are internally accessed using XPath syntax, users are not concerned with this as they always read or write whole CPOs. Once read, the CPO fields are contained in the (language specific) structure in memory.

Cpopath : within a dataset, a CPO is defined by the two elements (name, occurrence). The path of a CPO must be specified to the UAL as : ***<cpo name>/<occurrence number>.*** For example, *equilibrium/1* represents the path name of the first occurrence of the equilibrium CPO in the dataset. Note that the cpoPathName "equilibrium" is allowed and refers to a sort of 0^{th} occurrence of the CPO, though it should not be used by ITM programs under the KEPLER framework.

Fieldpath : in addition to Cpopath, path of the CPO fields must be specified to functions of the low level UAL (which operate on CPO fields). The field path uses also the Xpath syntax and is given relative to the CPO root. For example, the path of the q profile under a CPO equilibrium should be given as *profiles_1d/q*.

2.5 Empty fields

The ITM data structure aims at a fairly detailed description of the physics objects. Therefore it will be common that many fields are left empty by the program. UAL ***get()*** and ***put()*** routines will manage this transparently.

2.6 Memory Mapped Data Access

When running extensive simulation programs, the time required for I/O may worsen overall performance, especially when a large number of time values is managed and therefore a large number of CPO samples is appended one by one in the simulation database during execution. In order to improve (often dramatically) performance, the UAL allows to temporarily map CPO samples in memory, flushing them in the database under the control of the program.
When memory caching is enabled, read operations will access the simulation database once, maintaining afterwards the values in memory. Similarly, ***put()*** and ***putSlice()*** operations will store data in memory, until if is written back in the simulation database via a ***flush()*** operation.
Currently, the UAL allows globally enabling memory caching. It is then user's responsibility to flush CPO data from memory to disk. If data are not flushed, they are lost once the simulation database is closed.

2.7 UAL architecture

In order to cope with multiple languages and maintaining at the same time a unique structure definition, the UAL architecture defines two layers. The top layer provides the external Application Programming Interface (API), and its code is automatically produced from the XML description of the ITM database structure. For each supported programming language, a high level layer is generated in the target language. The following sections will describe the language specific API, and they provide all the required information for simulation program developers.
The lower layer is implemented in C and provides unstructured data access to the underlying database. It defines an API which is used by all the high level layer implementations. Knowledge of this API (presented in a later section) is not necessary to end users, and is only required to the

developers of new language specific high level implementations of the UAL as well as the developers of support tools for ITM management.
The definition of a low level API as the unique interface to low level I/O allows also the replacement of the underlying database implementation without affecting the high level layers.
The low level library provides support for MDSplus (the default configuration) and HDF5. The selection of the database technology is done in the ***open()*** or ***create()*** operations, and it is possible to handle at the same time MDSplus pulse files and HDF5 files. Knowledge of neither MDSplus nor HDF5 is required to operate the UAL.

3. High Level API

This section describes the language specific interface of the UAL for end users. Although all the implementations share the same concepts, they differ from each other, as the consequence of the differences among the programming languages.

3.1 FORTRAN Interface

For every CPO structure definition, a data type is defined in the Fortran interface. In order to make the compiler aware of the CPO data types and be able to use the Fortran UAL, it is necessary to include the following two lines in every Fortran program :

```
use euITM_schemas        ! ITM data types
use euITM_routines       ! Fortran UAL
```

New ITM simulation databases are created via routine

`euitm_create(treename,shot,run,refshot,refrun,idx)`

where:
- `treename` is the name of the ITM database, i.e. 'euitm' (mandatory);
- `shot, run` are the shot and run number of the database being created;
- `refshot, refrun` are the shot and run number of the reference database (currently not used)
- `idx` is the returned identifier to be used in the subsequent data access operation.

Existing simulation databases are open via routine

`euitm_open(treename,shot,run, idx)`

where:
- `treename` is the name of the ITM database, i.e. 'euitm' (mandatory);
- `shot, run` are the shot and run number of the database being created;
- `idx` is the returned integer identifier to be used in the subsequent data access operation.

Multiple datasets of the ITM database can be open at the same time. The returned `idx` parameter is used to re-direct I/O accordingly.

An open database is closed via routine:

`euitm_close(idx)`

where `idx` is the parameter returned by the open or create routine.

In order to read a CPO in a single get operation, it is necessary to declare first a variable whose type corresponds to the selected CPO structure. By definition the names of the CPO types in Fortran is ***type_<CPO name>***. For Time-independent CPOs, the declaration is straightforward. For example, the following declaration defines a variable `cpotest` to be of type `type_vessel`, where vessel is a Time-independent CPO.:

```
type (type_vessel) :: cpotest
```

For Time-dependent CPOs, get() operation may return an array of CPO samples. It is therefore necessary to declare a variable as a pointer (initialized to null) to the corresponding structure. In the following example, variable `cpotest1` is declared a pointer to type `type_equilibrium`, and will contain the returned array of equilibrium CPO samples:

```
type (type_equilibrium),pointer :: cpotest1(:) => null()
```

the following routine will get either a time-independent CPO or an array of time dependent CPO samples (multiple time slices) :

`euitm_get(idx,"Cpopath",retcpo)`

where:

- `idx` is the identifier returned by open or create;
- `Cpopath` is the (name,occurrence) of the CPO (e.g. “equilibrium” or “equilibrium/1”, see 2.4)
- `retcpo` is the returned CPO or CPO array

After `euitm_get` has been successfully executed, the array of CPO samples is contained in the returned `retcpo` variable. The number of returned samples (time slices) can be inspected by calling function `size(retcpo),` and the CPO fields are specified by the CPO occurrence number and the field name (the same as in the XML definition of the CPO structure). For example, field `datainfo/dataprovider` of the first time slice of a returned equilibrium CPO array (`cpotest`) is identified by:

```
cpotest(1)%datainfo%dataprovider
```

the following routine will return a single time slice of a time-dependent CPO. If applied (by mistake) to a time-independent CPO, it will simply call the **euitm_get** routine.

`euitm_get_slice(idx,"Cpopath",retcpo,twant,interpol)`

where:

- `idx` is the identifier returned by open or create;
- `Cpopath` is the (name,occurrence) of the CPO (e.g. “equilibrium” or “equilibrium/1”, see 2.4)
- `retcpo` is the returned CPO time slice. It is NOT an array.
- `twant` is the time value wanted by the user
- `interpol` is the interpolation mode (1,2 or 3) as defined in section 2.3

For a proper deallocation of the returned CPO structures the following routine should be used:

`euitm_deallocate(cpovar)`

where

-`cpovar` is the variable used to receive CPO samples in `euitm_get()`

CPOs already filled by the physics program can be written in the simulation database via routine:

euitm_put(idx,"Cpopath",cpovar)

where:

- idx is the identifier returned by either create or open;
- Cpopath is the (name,occurrence) of the CPO (e.g. “equilibrium” or “equilibrium/1”, see 2.4)
- cpovar is the variable holding the CPO. If the CPO is defined as time-dependent in the ITM data structure, cpovar must be an array of CPO samples.

Note that this routine will in first place erase any values that could have been written to this CPO before.

The following instruction will append a time slice to a CPO already put in the database. It exists only for time-dependent CPOs.

euitm_put_slice(idx,"Cpopath",cpovar)

where:

- idx is the identifier returned by either create or open;
- Cpopath is the (name,occurrence) of the CPO to append (e.g. “equilibrium” or “equilibrium/1”, see 2.4)
- cpovar is the variable holding the CPO time slice. It is NOT an array. Note that the time value of the slice is directly read in cpovar%time.

The following instruction will write only the time-independent fields of a CPO. It is useful to initialise a sequence of put_slice routines in a workflow (see 2.3). If applied (by mistake) to time-independent CPOs, it will call the ***euitm_put()*** instruction

euitm_put_non_timed(idx,"Cpopath",cpovar)

where:

- idx is the identifier returned by either create or open;
- Cpopath is the (name,occurrence) of the CPO (e.g. “equilibrium” or “equilibrium/1”, see 2.4)
- cpovar is the variable holding the CPO. If the CPO is defined as time-dependent in the ITM data structure, cpovar must be an array of CPO samples.

Note that this routine will in first place erase any values that could have been written to this CPO before.

The following instruction will delete entirely a CPO from the dataset. It is called by the euitm_put and euitm_put_non_timed routines systematically in order to clean completely the CPO prior any put.

euitm_delete(idx,"Cpopath")

where:

- idx is the identifier returned by either create or open;

- `Cpopath` is the (name,occurrence) of the CPO to delete (e.g. “equilibrium” or “equilibrium/1”, see 2.4)

Functions that are not yet implemented in the Fortran UAL :

`replaceLastSlice()`

`CPOcopy()`

The following routine enables data caching in memory:

`euitm_enable_mem_cache(shared, size)`

where:

- `shared` (boolean) specifies whether the memory cache is shared with other processes;
- `size` specifies the size of the data cache in memory. If the memory is insufficient, following put operations may fail.

When data caching is enabled, data in memory for a given CPO (or CPO array) is flushed on disk by calling the following routine:

`euitm_flush_cache(“CPO_name”)`

where:

- `CPO_name` is the path name of the CPO to be flushed in the simulation database.

Error handling in Fortran UAL

Allocating CPOs and fields

Note that all the CPO fields which are not a scalar value (i.e. strings and arrays) must be allocated prior to be used. If the CPO is returned by `euitm_get()`, the allocation is done by the UAL.

For example, the following instruction allocate an array of two time slices of a CPO, whose pointer is held in variable cpotest:

```
allocate(cpotest(2))
```

The following lines allocate field `profiles_1d/pprime`, which will contain an array of three float values, for both the allocated CPO time slices. **Note a CPO field must be allocated with the same dimension for all time slices of the CPO.**

```
do i=1,2
    allocate(cpotest(i)%profiles_1d%pprime(3)
enddo
```

It is also mandatory to fill the "time" field of every time-dependent CPO samples prior to inserting it in the simulation database.

Empty fields in Fortran :
Empty fields in Fortran CPOs are managed as follows :

- Empty integer fields are initialised as -999999999
- Empty real fields are initialised as -9.D40
- Empty pointers (strings and arrays) are initialised as « null ». During a program, there non-emptiness can be tested by the instruction « if (associated(varname)) »

Strings in Fortran :
Strings (XML type : xs:string) and arrays of strings (XML type : vecstring_type) are declared in F90 as : character(len=132), dimension(:), pointer

The proposed used is the following :

For strings :
The dimension of the F90 pointer must be allocated depending on the length of the string (i.e. more than 1 if the string length exceeds 132 characters). This allows to handle arbitrarily long strings. In the database, strings are stored as one arbitrarily long string. The F90 UAL does the concatenation of the F90 strings to the database format (or conversely the de-concatenation from the database to the F90).
Most strings will likely be short enough to be allocated with pointer dimension 1.

For vector of strings :
The dimension of the F90 pointer represents directly the dimension of the vector of strings. Each string in the vector must be of maximum length 132.

Note that at the moment, no time-dependent strings nor vector of strings are allowed in the schemas.

3.1.1 A FORTRAN example for reading an array of CPO samples

```
program test

! This program is for practicing the UAL GET command
! euitm_get gets all time slices of a CPO, replacing any previous data for that
CPO
! it can be used both for time-dependent and time-independent CPOs

use euITM_schemas
use euITM_routines

implicit none

integer,parameter :: DP=kind(1.0D0)


type (type_equilibrium),pointer :: cpotest(:) => null()
! For time-independent CPOs :
type (type_vessel) :: cpotest2

real(DP) :: scalans
character(len=132)::stringans,stringans2


integer :: idx, shot, run, status
```

```
integer :: numDims,dim1,dim2,dim3

character(len=5)::treename
shot =11 ! Your choice
run = 1  ! Your choice
treename = 'euitm'   ! Mandatory

 write(*,*) 'Open shot in MDS !'
 call euitm_open(treename,shot,run,idx)

write(*,*) 'Reading the results :'
call euitm_get(idx,"equilibrium",cpotest)    ! This does the memory allocation
automatically
write(*,*) 'This CPO has ',size(cpotest),' time slices'   ! check the number of
time slices present in the CPO

! Printout a few variables, just to show how it works
write(*,*) 'datainfo%provider = ',cpotest(1)%datainfo%dataprovider
write(*,*) 'codeparam%codename = ',cpotest(1)%codeparam%codename

write(*,*) 'time at time slice 1 = ',cpotest(1)%time
write(*,*) 'time at time slice 2 = ',cpotest(2)%time

write(*,*) 'li = ',cpotest(:)%global_param%li

write(*,*) 'Profiles_1d%pprime @ time slice 1 = ',cpotest(1)%Profiles_1d%pprime
write(*,*) 'Profiles_1d%pprime @ time slice 2 = ',cpotest(2)%Profiles_1d%pprime

write(*,*)      'Profiles_2d%psi_grid      @      time      slice      1      =
',cpotest(1)%Profiles_2d%psi_grid
write(*,*)      'Profiles_2d%psi_grid      @      time      slice      2      =
',cpotest(2)%Profiles_2d%psi_grid

call euitm_deallocate(cpotest) ! For a clean deallocation of the CPO variable
end
```

3.1.2 A FORTRAN example for writing an array of CPO samples

```
program test

! This program puts dummy data in the DB entry, just for practicing the UAL PUT
command
! euitm_put writes all time slices of a CPO, replacing any previous data for
that CPO
! it can be used both for time-dependent and time-independent CPOs

use euITM_schemas
use euITM_routines

implicit none

integer,parameter :: DP=kind(1.0D0)

type (type_equilibrium),pointer :: cpotest(:) => null() ! Declaration of the
empty CPO to be filled
! This is a pointer because it is time dependent (several time slices allowed)
and we want it to have several time slices
! In the case of a time-independent CPO or a single time slice of a time
dependent CPO, one should declare it as :
! type (type_vessel) :: cpotest


integer :: idx, shot, run, refshot, refrun, status, i

character(len=5)::treename
shot =11
run = 1
refshot = 10
refrun =0
treename = 'euitm'

 write(*,*) 'Open shot in MDS !'
 call euitm_open(treename,shot,run,idx)

 ! Let's do a CPO with 2 time slices
 allocate(cpotest(2))

! Filling part of the tree

 allocate(cpotest(1)%datainfo%dataprovider(1))  ! Example for a string variable.
Attention : all strings are allocatable !!
 cpotest(1)%datainfo%dataprovider="FI"            ! NB : this data is NOT time-
dependent, though the CPO is time-dependent. The euitm_put instruction will use
only its value as defined for the first time slice, i.e. in cpotest(1)

 allocate(cpotest(1)%codeparam%codename(3))   ! Example for an array of strings
(the length of each string of the array is limited to 132 characters)
 cpotest(1)%codeparam%codename(1)="test_equilibrium_put.f90 : First line"
 cpotest(1)%codeparam%codename(2)="test_equilibrium_put.f90 : Second line"
 cpotest(1)%codeparam%codename(3)="test_equilibrium_put.f90 : Third line"


 cpotest(1)%global_param%li  = 1.0D0       ! Example of a time-dependent scalar
variable
 cpotest(2)%global_param%li = 2.0D0

 ! Example of a time-dependent vector variable
 do i=1,2
    allocate(cpotest(i)%profiles_1d%pprime(3))     ! All time slices must have
the same dimension !!
 enddo
 cpotest(1)%profiles_1d%pprime= (/ 3.2D0 , 4.2D0, 5.2D0 /)
 cpotest(2)%profiles_1d%pprime= (/ 5.2D0 , 6.2D0, 7.2D0 /)

 ! Example of a time-dependent matrix variable
 do i=1,2
```

```
    allocate(cpotest(i)%profiles_2d%psi_grid(4,3)) ! All time slices must have
the same dimension !!
 enddo
 cpotest(1)%profiles_2d%psi_grid(1,:)= (/ 3.2D0 , 4.2D0, 5.2D0 /)
 cpotest(1)%profiles_2d%psi_grid(2,:)=(/ 3.3D0 , 4.3D0, 5.3D0 /)
 cpotest(1)%profiles_2d%psi_grid(3,:)=(/ 3.4D0 , 4.4D0, 5.4D0 /)
 cpotest(1)%profiles_2d%psi_grid(4,:)=(/ 3.5D0 , 4.5D0, 5.5D0 /)
 cpotest(2)%profiles_2d%psi_grid(:,:)                                        =
cpotest(1)%profiles_2d%psi_grid(:,:)+1.0D0

 cpotest(1:2)%time= (/ 1.2D0 , 2.2D0/) ! IT IS MANDATORY TO DEFINE THE TIME
VARIABLE IN A TIME-DEPENDENT CPO

 write(*,*) 'Put full Equilibrium CPO'
 call euitm_put(idx,"equilibrium",cpotest)

 call euitm_deallocate(cpotest)

end
```

To be added : a couple of more examples for reading and writing CPO slices

---

3.2 C++ Interface

In C++ the UAL functionality is exported by a set of classes. All the required classes are defined in `UALClasses.h`, which has to be included in the program.

All the UAL classes and methods are defined in namespace `ItmNs`. The UAL C++ interface uses the **blitz** package for handling arrays. Refer to http://www.oonumerics.org/blitz/ for the blitz related documentation.

The main class for the UAL is `ItmNs::Itm`. Its constructor takes 4 integer parameters: `Shot, Run, RefShot, RefRun` (the last two arguments are currently not used).
The following methods are defined for class ItmNs::Itm:

- **`void open()`** Open the simutation database;
- **`void create()`** Creates a new simulation database;
- **`void close()`** Closes the simulation database;
- **`bool isConnected()`** Return true if a simulation database is currently open.

The last method should be called in order to make sure that `open()` or `create()` were successful.
Method `close()` is also called by the destructor of class `ItmNs::Itm`.

An object of class `ItmNs::Itm` contains a set of fields which correspond to the CPO types defined in the ITM simulation database. For Time-independent CPOs, an inner class with the same name is defined, and a field of the `ItmNs::Itm` is declared, whose name is the name of the CPO type preceded by an underscore. For example, if `itm` is a variable of class `ItmNs::Itm`, `itm._summary` (of class `ItmNs::Itm::summary`) contains all the fields of the ***summary*** CPO. CPO fields are then accessed as fields of the corresponding class. For example, the string field whose path in the XML definition is `summary/datainfo/comment`, is accessed in C++ as `itm._summary.datainfo.comment` and is represented by C++ `string` type. Scalar CPO fields are mapped to the corresponding C++ types: `int` for integer fields, `float` for float

fields and `double` for double fields. Mono and multidimensional arrays are instead represented by blitz array objects, thus allowing a very flexible and efficient array management.

For time-independent CPOs, the corresponding class defines the following methods:

- **`int get()`** Read the time-independent CPO in the simulation database
- **`int get(int inst)`** Read the time-independent CPO whose instance number is given by argument `inst` (to be used if multiple CPO instances are used);
- **`int put()`** Write the time-independent CPO in the simulation database.
- **`int put(int inst)`** Write the time-independent CPO whose instance number is given by argument `inst` (to be used if multiple CPO instances are used);

For example, the summary CPO is read in the database in the following instruction (where `itm` is an instance of class `ItmNs::Itm`, for which method `open()` has been successfully called):

```
itm._summary.get();
```

Upon successful completion the above methods return 0. If a different value is returned, an error occurred. Routine

`char *euitm_last_errmsg()`

will return a string description of the error which occurred last.

For time-dependent CPOs, two inner classes and two fields in class ItmNs::Itm are defined. The first class has the same name of the CPO type, and the corresponding field has the same name, preceded by an underscore. The second class has name `<CPO type>Array` and the corresponding field has the class name preceded by an underscore.
The first class (and field) is used for all those operation which involve a single CPO sample, that is getSlice() and putSlice(). The second class will instead andle all the operation which involve arrays of CPO samples, i.e. get() and put().

The methods defined in the first class are the following:

- **`int getSlice(double inTime, char interpolMode)`** Read this CPO from the simulation database, at the time corresponding to `inTime`, using the interpolation mode specified in `interpolMode`;
- **`int getSlice(int inst, double inTime, char interpolMode)`** Same as before, but for the CPO instance specified by argument `inst`;
- **`int putSlice()`** Append this CPO to the CPO array stored in the simulation database. Only time-dependent CPO fields will be written in the database, and the time field of this CPO will specify the time this CPO instance refers to;
- **`int putSlice(int inst)`** Same as before, but for the CPO instance specified by argument `inst`;
- **`int replaceLastSlice()`** Replace the last CPO in the CPO array stored in the simulation database;
- **`int replaceLastSlice(int inst)`** Same as before, but for the CPO instance specified by argument `inst`;
- **`int putNonTimed()`** Write in the database the time-independent fields of this CPO
- **`int putNonTimed(int inst)`** Same as before, but for the CPO instance specified by argument inst;

- **`void flushCache()`** Flush all data cached in memory for this CPO into the simulation database.

The definition of two different methods for storing time-dependent and time-independent CPO fields allows improving performance. Time-independent CPO fields are in fact required to be stored only once in the database, while time-dependent fields need to be appended in the database as many times as the number of time samples computed in the simulation.

The second class is instead used when arrays of CPO samples are handled by the program. i.e for get() and put() operation for time-dependent CPOs. The name of this class is the <CPO type>Array, and defines an unique field named `array` which is a blitz array of <CPO types> objects (i.e. an array of objects of the first class). The methods defined for this class are the following:

- **`int get()`** Read the stored array of CPOs in the simulation database;
- **`int get(int inst)`** Same as before, but for the CPO instance specified by argument inst;
- **`int put()`** Write in the simulation database the array of CPO samples stored in the array field;
- **`int put(int idx)`** Same as before, but for the CPO instance specified by argument inst;

Once the array of CPO samples has been read from the simulation database, the fields of the ith CPO sample are accessed as the ith element of the field array. For example, assuming that `itm` is an instance of class `ItmNs::Itm`, the number of **pfsystems** CPO samples read in the simulation database, i.e. after a successful execution of `itm._pfsystemsArray.get()`, is given by:

```
itm._pfsystemsArray.array.extent(0)
```

and the kth element of the CPO field (a double array) whose path name is pfsupplies/voltage/value of the ith CPO samples is given by:

```
itm._pfsystemsArray.array(i).pfsupplies.voltage.value(k)
```

Empty fields in C++ CPOs are managed as follows :

- Empty integer fields are initialised as -999999999
- Empty real fields are initialised as -9.D40
- Empty pointers (strings and arrays) are initialised as 0. It is responsibility of the programmer to test for the existence of arrays before using blitz accessor methods.

3.2.1 A C++ example for reading a CPO slice

```
//Definition of the class structures in file UALClasses.h
#include "UALClasses.h"
/*
 This sample program reads the database filled by program put_cpo_slice_cxx. It takes a
time argument
from the command line, and uses this value to get a time slice in the array of stored
pfsystems CPOs.
Linear interpolation is [performed if the time lies beween stored times. It the time lies
outsize the
range of stored times, the upper or lower CPO slice is considered.
*/

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

```
{
    float time;
    if(argc != 2)
    {
        printf("Usage: get_cpo_clice_cxx <time>\n");
        exit(0);
    }
   sscanf(argv[1], "%f", &time);

    //Here refShot and refRun arguments have no meaning
    ItmNs::Itm itm(123,2,123,1);
    itm.open(); //Open the database


    //Get the  CPO slice corresponding to the passed time
    //Defining linear interpolation. The other options are CLOSEST_SAMPLE
    //and PREVIOUS_SAMPLE
    itm._pfsystems.getSlice(time, INTERPOLATION);

    //Dump the whole returned CPO
      cout << "PFSYSTEMS at time " << time << "\n" << itm._pfsystems;
}
```

3.2.2 A C++ example for writing a CPO slice

```
//Definition of the class structures in file UALClasses.h
#include "UALClasses.h"


/*
This programs append 20 pfsystems CPO slices for pfsystems CPO. The usage of this
approach makes sense during a simulation in which subsequent
time slices are computed and then appended in the database. */

int main(int argc, char *argv[])
{
    //Instantiate the Itm object for shot 123, run 2, using shot 123 and
    //run 0 as reference shot (not used for now)
        ItmNs::Itm itm(123,2,123,0);

        //Create a new instance of database
        itm.create();


        //First fill fields which are not time-dependent. This is carried out
         // by method putNonTimed()


        //Fill a string field (datainfo.dataprovider) Strings are represented
        //by std::string objects

        itm._pfsystems.datainfo.dataprovider.assign("USER");

        //Allocate and fill with sample values field
        //pfcoils.desc_pfcoils.res (1D float array)

        itm._pfsystems.pfcoils.desc_pfcoils.res.resize(4);
        for(int i = 0; i < 4; i++)
            itm._pfsystems.pfcoils.desc_pfcoils.res(i) = i;


        //Allocate and fill field pfcoils.desc_pfcoils.name (1D String array)

        itm._pfsystems.pfcoils.desc_pfcoils.name.resize(2);
        itm._pfsystems.pfcoils.desc_pfcoils.name(0) = "SAMPLE 1";
        itm._pfsystems.pfcoils.desc_pfcoils.name(1) = "SAMPLE 2";

        //Allocate and fill field
        //pfcoils.desc_pfcoils.pfelement.pfgeometry.rzcoordinate.r
         // (3D float array).

itm._pfsystems.pfcoils.desc_pfcoils.pfelement.pfgeometry.rzcoordinate.r.resize(2,3,4);
        for(int i=0 ; i < 2; i++)
            for(int j = 0; j < 3; j++)
                for(int h = 0; h < 4; h++)
```

```
itm._pfsystems.pfcoils.desc_pfcoils.pfelement.pfgeometry.rzcoordinate.r(i,j,h)          =
100*i+10*j+h;

        //Save non time dependent fields
        itm._pfsystems.putNonTimed();

        //Save the time evolution of time dependent fields (only pfsupplies.voltage.value
        // is fileld in this example)

        //Resize first (and once) the array. Recall that pfsupplies.voltage.value is a
        // 1D time dependent array
        itm._pfsystems.pfsupplies.voltage.value.resize(4);
        for(int k = 0; k < 200; k++)
        {
            for(int i = 0; i < 4; i++)
                itm._pfsystems.pfsupplies.voltage.value(i) = 111*(i+1)+1000*k;

            //Do not forget time!!
            itm._pfsystems.time = k;

            //Append this slice in the database
            itm._pfsystems.putSlice();
        }
        //Done, let the destructor close the database
}
```

3.2.3 A C++ example for reading a whole CPO array

```
//Definition of the class structures in file UALClasses.h
#include "UALClasses.h"

/*
This sample program will open an eisting pulse file (shot 123, run1) and will
read the stored (array of) equilibirium CPOs.
It will then output the content of some fields of the equilibrium CPOs.
*/

int main(int argc, char *argv[])
{
    //Class Itm is the main class for the UAL. It contains a set of field classes,
    //each corresponding to a CPO defined in the UAL
    //The parameters passed to this creator define the shot and run number.
    //The second pair of arguments defines the reference shot and run
    //and is used when the a new database is created, as in this example.
    //All the UAL classes belong to the ItmNs namespace

    ItmNs::Itm itm(123,1,123,0);

    itm.open(); //Open the database

    //Read the whole array of pfsupplies CPOs
     itm._pfsystemsArray.get();

     //field array of inner class _equilbriumArray is a blitz array containing
     // the array of CPOs
     //(objects of class _equilibrium)
     cout << "Method get() for pfsupplies CPO returned " <<
       itm._pfsystemsArray.array.extent(0) << " CPO instances\n";

     //Print the contents of fields time and pfsupplies.voltage.value
     for(int i = 0; i < itm._pfsystemsArray.array.extent(0); i++)
     {
         cout << "\nCPO " << i << ": time = " << itm._pfsystemsArray.array(i).time <<
        "profiles_1d.F_dia = " << itm._pfsystemsArray.array(i).pfsupplies.voltage.value
        <<"\n";
     }

     //Dump of the whole CPO array
     cout << itm._pfsystemsArray.array;
```

```
     //Dump only the first slice of the CPO array
     cout << itm._pfsystemsArray.array(0);
}
```

3.2.4 A C++ example for writing a whole CPO array

```
//Definition of the class structures in file UALClasses.h
#include "UALClasses.h"
/*
This sample program will create a new pulse file (shot 123, run 1), and then will fill
two CPOs in in it: equilibrium and and pfsystems. For every CPO three time slices are
considered. An array of three CPO
 instances will be created for both equilibrium and pfsystems.
*/

int main(int argc, char *argv[])
{
    //Class Itm is the main class for the UAL. It contains a set of field classes,
    //each corresponding to a CPO defined in the UAL
    //The parameters passed to this creator define the shot and run number.
    //The second pair of arguments defines the reference shot and run
    //and is used when the a new database is created, as in this example.
    //All the UAL classes belong to the ItmNs namespace

    ItmNs::Itm itm(123,1,123,0);

    itm.create(); //Create a new instance of database

    //We shall save an array of pfsystems and equilibrium CPOs. These CPO are declared
    //in the UAL as time-dependent, i.e. they contain
    //one field which refers to a value depending over time.
    //For time-dependent CPOs the field "time" is always defined and MUST be filled with
    //the value of the time the CPO instance refers to
    //Observe that there are a few CPOs which are not time dependent. For those CPOs
    //(e.g. summary) only methods put() and get() are defined.
    //For every time dependent CPO, two classes are defined. The first one is named
    // <CPO type>, and contains all the fields which
    //are defined in the CPO structure. The second class is named <CPO name>Array and is
    //defined to contain an array of _<CPO name> objects.
    //Class <CPO name>Array is used when dealing with arrays of CPOs, i.e. when using
    //methods put()and get()
    //Class <CPO name> is used when dealing with a single CPO instance, i.e. when using
    //methods putNonTimed(), putSlice() and getSlice().
    //In this example we are going to write in the database an array of pfsystems CPOs
    //and an array of equilibrium
    //CPOs, so we shall use class _pfsystemsArray and _equilibriumArray.
    //Observe that all the CPO classes are inner class of class Itm.

    //All arrays here are blitz arrays.

    //Allocate room for 3 CPO instances
    itm._pfsystemsArray.array.resize(3);
    itm._equilibriumArray.array.resize(3);

    //Fill some fields of the CPO array instances
    for(int k = 0; k < 3; k++)
    {
            //Fill a string field (datainfo.dataprovider) Strings are represented
            //by std::string objects

            itm._pfsystemsArray.array(k).datainfo.dataprovider.assign("GAB");
            itm._equilibriumArray.array(k).codeparam.parameters.assign("xmltoken");

            //The following fields are not time dependent, so we put the same value for
             //every CPO instance.
            //Note that the UAL refer only to the first instance when saving non time
            //dependent field, so
            //it would suffices to store non time dependent field for
           //itm._pfsystemsArray.array(0).

            //Allocate and fill with sample values field pfcoils.desc_pfcoils.res
           //(1D float array)

            itm._pfsystemsArray.array(k).pfcoils.desc_pfcoils.res.resize(4);
            itm._equilibriumArray.array(k).profiles_1d.elongation.resize(4);
            for(int i = 0; i < 4; i++)
            {
                itm._pfsystemsArray.array(k).pfcoils.desc_pfcoils.res(i) = i;
```

```
                itm._equilibriumArray.array(k).profiles_1d.elongation(i) = i;
            }

            //Allocate and fill field pfcoils.desc_pfcoils.name (1D String array)
            itm._pfsystemsArray.array(k).pfcoils.desc_pfcoils.name.resize(2);
            itm._pfsystemsArray.array(k).pfcoils.desc_pfcoils.name(0) = "SAMPLE 1";
            itm._pfsystemsArray.array(k).pfcoils.desc_pfcoils.name(1) = "SAMPLE 2";


            //Allocate and fill field
             //pfcoils.desc_pfcoils.pfelement.pfgeometry.rzcoordinate.r (3D float array).

itm._pfsystemsArray.array(k).pfcoils.desc_pfcoils.pfelement.pfgeometry.rzcoordinate.r.resize(2,3,4);
            for(int i=0 ; i < 2; i++)
                for(int j = 0; j < 3; j++)
                    for(int h = 0; h < 4; h++)

itm._pfsystemsArray.array(k).pfcoils.desc_pfcoils.pfelement.pfgeometry.rzcoordinate.r(i,j,h)
       = 100*i+10*j+h;



            //The following field is time dependent. We shall put some values which
depend on the time value k.
            //pfsupplies.voltage.value is a 1D time dependent float vector
            itm._pfsystemsArray.array(k).pfsupplies.voltage.value.resize(4);
            for(int i = 0; i < 4; i++)
                itm._pfsystemsArray.array(k).pfsupplies.voltage.value(i) =
                    111*(i+1)+1000*k;


            //Never forget to store the time for time dependent CPOs!!
            itm._pfsystemsArray.array(k).time = k;
            itm._equilibriumArray.array(k).time = k;
    }
    //Now the CPO array is filled, we store in the database via method put()
    itm._pfsystemsArray.put();
    itm._equilibriumArray.put();
    //There is no need to explicitly close the database since it is closed by the
    //destructor of class Itm.
}
```

3.3 UAL Python Interface

The ual interface for Python is very close to the C++ one. Only namespace definition and some languages specificities differs.

Declaration

The UAL functionality is exported by a set of classes which can be accessed in Python by writing

```
import ual
```

Namespace usage in Python, and field names

All the UAL classes and methods are defined in the ual package. The UAL Python interface uses the numpy package for handling arrays. Refer to the numpy related documentation for more information.

The main class for the UAL is ual.itm. It contains a set of fields which correspond to the CPO types defined in the ITM simulation database. For Time-independent CPOs, an inner class with the same name is defined, and a field of the ual.itm class is declared, whose name is the name of the CPO type. For example, if my_itm_obj is a variable of class ual.itm, itm.summary contains all the fields of the summary CPO. CPO fields are then accessed as fields of the corresponding class. For example, the string field whose path in the XML definition is summary/datainfo/comment, is accessed in Python as itm.summary.datainfo.comment and is represented by Python string type.

Scalar CPO fields are mapped to the corresponding Python types: int for integer fields and double for double and float fields. Mono and multidimensional arrays are instead represented by numpy.ndarray objects, thus allowing a very flexible and efficient array management.

Constructor of ual.itm takes 4 integer parameters: Shot, Run, RefShot, RefRun (the last two arguments are currently not used).

```
my_itm_obj = ual.itm(123,1,123,0)
```

Simulation database creation/open

The following methods are defined for class ual.itm

```
void open()
```

- Open the simulation database

```
void create()
```

- Creates a new simulation database

```
void close()
```

- Closes the simulation database

```
bool isConnected()
```

- Return true if a simulation database is currently open

The last method should be called in order to make sure that open() or create() were successful. Method close() is also called by the destructor.

Read/write for time-independent CPOs

All these methods return an integer which tells if an error occured. Each of these methods takes an optional argument which is the occurence number. If a single occurence of a CPO is used, no argument must be given. Normally a python method definition does not contain the type of the arguments. But for the sake of clarity, type of awaited arguments are given using a C convention.

```
int get(int occ=-1)
```

- Read the time-independent CPO in the simulation database

```
int put(int occ=-1)
```

- Write the time-independent CPO in the simulation database

For example, the summary CPO is read in the database in the following instruction. my_itm_obj is an instance of class ual.itm, for which method open() has been successfully called:

```
my_itm_obj.summary.get()
```

Upon successful completion the above methods return 0. If a different value is returned, an error occurred. Routine char *euitm_last_errmsg() will return a string description of the error which occurred last.

Read/write for time-dependent CPOs

For time-dependent CPOs, two inner classes and two fields in class ual.itm are defined. The first class has the same name of the CPO type, and the corresponding field has the same name. The

second class has name <CPO type>Array and the corresponding field has the name. The first class (and field) is used for all those operation which involve a single CPO sample, that is getSlice() and putSlice(). The second class will instead handle all the operation which involve arrays of CPO samples, i.e. get() and put().

The methods defined in the first class are the following. They all return an integer which tells if an error occured. Each of these methods takes an optional argument which is the occurence number. If a single occurence of a CPO is used, no argument must be given. Normally a python method definition does not contain the type of the arguments. But for the sake of clarity, type of awaited arguments are given using a C convention.

```
int getSlice(int occ=-1, double inTime, char interpolMode)
```

- Read this CPO from the simulation database, at the time corresponding to inTime, using the interpolation mode specified in interpolMode

```
int putSlice(int occ=-1)
```

- Append this CPO to the CPO array stored in the simulation database. Only time-dependent CPO fields will be written in the database, and the time field of this CPO will specify the time this CPO instance refers to

```
int replaceLastSlice(int occ=-1)
```

- Replace the last CPO in the CPO array stored in the simulation database

```
int putNonTimed(int occ=-1)
```

- Write in the database the time-independent fields of this CPO

```
void flushCache()
```

- Flush all data cached in memory for this CPO into the simulation database.

The definition of two different methods for storing time-dependent and time-independent CPO fields allows improving performance. Time-independent CPO fields are in fact required to be stored only once in the database, while time-dependent fields need to be appended in the database as many times as the number of time samples computed in the simulation.

The second class is instead used when arrays of CPO samples are handled by the program. i.e for get() and put() operation for time-dependent CPOs. The name of this class is the <CPO type>Array, and defines an unique field named array which is a blitz array in C++ (resp. a numpy.ndarray in Python) of <CPO types> objects (i.e. an array of objects of the first class). The methods defined for this class are the following:

```
int get(int occ=-1)
```

- Read the stored array of CPOs in the simulation database

```
int put(int occ=-1)
```

- Write in the simulation database the array of CPO samples stored in the array field

```
int getResampled(int occ=-1, double start, double end, double step, char
interpolMode)
```

- Resample an array of CPO from the stored array of CPOs in the simulation database. <CPO type>Array object built will contain (end-start)/step time slices resampled from the original data by using the interpolation defined by interpolMode.

Once the array of CPO samples has been read from the simulation database, the fields of the ith CPO sample are accessed as the ith element of the field array.

For example, assuming that my_itm_obj is an instance of class ual.itm, the number of pfsystems CPO samples read in the simulation database, i.e. after a successful execution of my_itm_obj.pfsystemsArray.get(), is given by:

```
len(my_itm_obj.pfsystemsArray)
```

and the kth element of the CPO field (an array of double) whose path name is pfsupplies/voltage/value of the ith CPO samples is given by:

```
my_itm_obj.pfsystemsArray.array[i].pfsupplies.voltage.value[k]
```

Python example of putting a CPO

```
#Definition of the class structures in file ual.py
import ual
import numpy

'''
This sample program will create a new pulse file (shot 123, run 2), and then will fill two CPOs in
it:
equilibrium and and pfsystems. For every CPO three time slices are considered. An array of three CPO
 instances will be created for both equilibrium and pfsystems.
'''
def write_cpo():
        '''Class Itm is the main class for the UAL. It contains a set of field classes, each
corresponding to a CPO defined in the UAL
        The parameters passed to this creator define the shot and run number. The second pair of
arguments defines the reference shot and run
        and is used when the a new database is created, as in this example.
        '''

        my_itm_obj = ual.itm(123,3,123,0)

        my_itm_obj.create() #Create a new instance of database

        #We shall save an array of pfsystems and equilibrium CPOs. These CPO are declared in the UAL
as time-dependent, i.e. they contain
        #one field which refers to a value depending over time.
        #For time-dependent CPOs the field "time" is always defined and MUST be filled with the
value of the time the CPO instance refers to
        #Observe that there are a few CPOs which are not time dependent. For those CPOs (e.g.
summary) only methods put() and get() are defined.
        #For every time dependent CPO, two classes are defined. The first one is named <CPO name>,
and contains all the fields which
        #are defined in the CPO structure. The second class is named <CPO name>Array and is defined
as an array of <CPO name> objects.
        #Class <CPO name>Array is used when dealing with arrays of CPOs, i.e. when using methods
put(), get() and getResampled()
        #Class <CPO name> is used when dealing with a single CPO instance, i.e. when using methods
putNonTimed(), putSlice() and getSlicce().
        #In this example we are going to write in the database an array of pfsystems CPOs and an
array of equilibrium
        # CPOs, so we shall use class pfsystemsArray and equilibriumArray.
        #Observe that all the CPO classes are inner class of class Itm.

        #All arrays here are numpy.ndarray objects.

        #Allocate room for 3 CPO instances
        nb_cpos=3
        my_itm_obj.pfsystemsArray.resize(nb_cpos)
        my_itm_obj.equilibriumArray.resize(nb_cpos)

        #Fill some fields of the CPO arrayinstances
        for k in range(nb_cpos):
                #Fill a string field (datainfo.dataprovider) Strings are represented by std::string
objects
                my_itm_obj.pfsystemsArray.array[k].datainfo.dataprovider = 'GAB'
                my_itm_obj.pfsystemsArray.array[k].datainfo.putdate = '16/02/2009'
```

```
                my_itm_obj.equilibriumArray.array[k].codeparam.parameters = 'xmltoken'

                #The following fields are not time dependent, so we put the same value for every CPO
instance.
                #Note that the UAL refer only to the first instance when saving non time dependent
field, so in
                #it would suffices to store non time dependent field for
itm._pfsystemsArray.array(0).

                #Allocate and fill with sample values field pfcoils.desc_pfcoils.res (1D float
array)
                nb_values = 4
                res_loc = numpy.resize(my_itm_obj.pfsystemsArray.array[k].pfcoils.desc_pfcoils.res,
nb_values)
                elongation_loc =
numpy.resize(my_itm_obj.equilibriumArray.array[k].profiles_1d.elongation, nb_values)

                for i in range(nb_values):
                        res_loc[i] = i
                        elongation_loc[i] = i


                #No copy is performed, only a reference affectation
                my_itm_obj.pfsystemsArray.array[k].pfcoils.desc_pfcoils.res = res_loc
                my_itm_obj.equilibriumArray.array[k].profiles_1d.elongation = elongation_loc

                #Allocate and fill field pfcoils.desc_pfcoils.name (1D String array)
                my_itm_obj.pfsystemsArray.array[k].pfcoils.desc_pfcoils.name = ['SAMPLE 1', 'SAMPLE
2']
                my_itm_obj.pfsystemsArray.array[k].pfcoils.desc_pfcoils.id = ['ID 1', 'ID 2']

                #Allocate and fill field pfcoils.desc_pfcoils.pfelement.pfgeometry.rzcoordinate.r
                #(3D float array).
                r_loc = numpy.resize(
my_itm_obj.pfsystemsArray.array[k].pfcoils.desc_pfcoils.pfelement.pfgeometry.rzcoordinate.r,
(2,3,4))
                for i in range(2):
                        for j in range(3):
                                for h in range(4):
                                        r_loc[i,j,h] = 100.0*i+10.0*j+h


my_itm_obj.pfsystemsArray.array[k].pfcoils.desc_pfcoils.pfelement.pfgeometry.rzcoordinate.r = r_loc


                r_loc2 = numpy.resize(
my_itm_obj.pfsystemsArray.array[k].pfpassive.pfpageometry.rzcoordinate.r, (2,3))
                for i in range(2):
                        for j in range(3):
                                r_loc2[i,j] = 100.0*i+10.0*j

                my_itm_obj.pfsystemsArray.array[k].pfpassive.pfpageometry.rzcoordinate.r = r_loc2

                #The following field is time dependent. We shall put some values which depend on the
time value k.
                #pfsupplies.voltage.value is a 1D time dependent float vector
                value_loc =
numpy.resize(my_itm_obj.pfsystemsArray.array[k].pfsupplies.voltage.value, 10)
                for i in range(10):
                        value_loc[i] = 111.0*(i+1.0)+1000.0*k

                my_itm_obj.pfsystemsArray.array[k].pfsupplies.voltage.value = value_loc

                #Never forget to store the time for time dependent CPOs!!
                my_itm_obj.pfsystemsArray.array[k].time = k
                my_itm_obj.equilibriumArray.array[k].time = k


        #Dump all CPO and CPOArray of the itm object. Very verbose !!
        #print my_itm_obj

        #Dump only the first time slice of the pfsystemsArray CPO. Still verbose
        #print my_itm_obj.pfsystemsArray.array[0]

        #Dump only a part of the  first time slice of pfsystemsArray
        print my_itm_obj.pfsystemsArray.array[0].pfsupplies

        #Now the CPO array is filled, we store in the database via method put()
```

```
        my_itm_obj.pfsystemsArray.put()
        my_itm_obj.equilibriumArray.put()


        #There is no need to explicitly close the database since it is closed by
         #the destructor of class Itm.

        for i in range(len(my_itm_obj.pfsystemsArray.array)):
                print "CPO " + str(i) + ": time = " + str(my_itm_obj.pfsystemsArray.array[i].time) +
" profiles_1d.F_dia = " +  str(my_itm_obj.pfsystemsArray.array[i].pfsupplies.voltage.value)

#       print
my_itm_obj.pfsystemsArray.array[0].pfcoils.desc_pfcoils.pfelement.pfgeometry.rzcoordinate.r
#       print
my_itm_obj.pfsystemsArray.array[0].pfcoils.desc_pfcoils.pfelement.pfgeometry.rzcoordinate.r.flags
#       print
my_itm_obj.pfsystemsArray.array[0].pfcoils.desc_pfcoils.pfelement.pfgeometry.rzcoordinate.r.ravel()

write_cpo()
```

Python example of putting an array of CPO using putSlice mechanism

```
#Definition of the class structures in file ual.py
import ual
import numpy

'''
This programs append 20 pfsystems CPO slices for pfsystems CPO. Instead of
writing the whole CPO array in a single chunk, as done in program put_cpos.py, every slice is saved
(appended) individually. The usage of this approach makes sense during a simulation in which
subsequent
time slices are computed and then appended in the database.
Here, the inner class pfsystem and equilibrium if the ual.itm class instance are used, instead of
pfsystemsArray and equilibriumArray, as done in program put_cpos.py. In this case in fact we deal
with
a single CPO slice at any time
'''

def write_cpo():
        #Instantiate the Itm object for shot 123, run 2, using shot 123 amd run 0 as reference shot
        my_itm_obj = ual.itm(123,4,123,0)

        #Create a new instance of database
        my_itm_obj.create()

        #First fill fields which are not time-dependent. This is cassier out by method putNonTimed()

        #Fill a string field (datainfo.dataprovider) Strings are represented by std::string objects
        my_itm_obj.pfsystems.datainfo.dataprovider = "USER"

        #Allocate and fill with sample values field pfcoils.desc_pfcoils.res (1D float array)
        res_loc = numpy.resize(my_itm_obj.pfsystems.pfcoils.desc_pfcoils.res, 4)
        for i in range(4):
                res_loc[i] = i

        #No copy is performed, only a reference change in python
        my_itm_obj.pfsystems.pfcoils.desc_pfcoils.res = res_loc

        #Allocate and fill field pfcoils.desc_pfcoils.name (1D String array)
        my_itm_obj.pfsystems.pfcoils.desc_pfcoils.name = ["SAMPLE 1","SAMPLE 2"]

        #Allocate and fill field pfcoils.desc_pfcoils.pfelement.pfgeometry.rzcoordinate.r
         #(3D float array).
        r_loc =
numpy.resize(my_itm_obj.pfsystems.pfcoils.desc_pfcoils.pfelement.pfgeometry.rzcoordinate.r, (2,3,4))
        for i in range(2):
                for j in range(3):
                        for h in range(4):
                                r_loc[i,j,h] = 100.0*i+10.0*j+h

        my_itm_obj.pfsystems.pfcoils.desc_pfcoils.pfelement.pfgeometry.rzcoordinate.r = r_loc

        #Save non time dependent fields
        my_itm_obj.pfsystems.putNonTimed()

        #Save the time evolution of time dependent firlds
```

```
         #(only pfsupplies.voltage.value is fileld in this example)

        #Resize first (and once) the array. Recall that pfsupplies.voltage.value
       #is a 1D time dependent array
        value_loc = numpy.resize(my_itm_obj.pfsystems.pfsupplies.voltage.value, 4)
        my_itm_obj.pfsystems.pfsupplies.voltage.value = value_loc
        for k in range(20):

                for i in range(4):
                        value_loc[i] = 111.0*(i+1.0)+1000.0*k

                #Do not forget time!!
                my_itm_obj.pfsystems.time = k

                #Append this slice in the database
                my_itm_obj.pfsystems.putSlice()
        #Done, let the destructor close the database

write_cpo()
```

Python example to get a CPO array

```
#Definition of the class structures in file ual.py
import ual
import numpy
from ualdef import *

'''
This sample program will open an existing pulse file (shot 123, run 3, created by script
put_cpos.py) and will
read the stored (array of) equilibirium CPOs.

It will then output the content of some fields of the equilibrium CPOs.
'''

#This routine reads an array of pfsystems CPOs in the database, filling some fields of the CPOS
def read_cpo():
        '''Class Itm is the main class for the UAL. It contains a set of field classes, each
corresponding to a CPO defined in the UAL
        The parameters passed to this creator define the shot and run number. The second pair of
arguments defines the reference shot and run
        and is used when the a new database is created, as in this example.
        '''
        my_itm_obj = ual.itm(123,3,123,0)

        my_itm_obj.open() #Open the database

        #Read the whole array of pfsupplies CPOs
        my_itm_obj.pfsystemsArray.get()

        #Read a resampled version of the pfsystem
        #my_itm_obj.pfsystemsArray.getResampled(5.0,10.0,0.5,INTERPOLATION)


        #field array of inner class equilbriumArray is a list containing the array of CPOs
        #(objects of class pfsystems)
        print "Method get() for pfsupplies CPO returned " +
str(len(my_itm_obj.pfsystemsArray.array)) + " CPO instances\n";

#       print 'Dump of the whole CPO array'
#       print my_itm_obj.pfsystemsArray.array


#       print 'Dump only the first slice of the CPO array'
#       print my_itm_obj.pfsystemsArray.array[0]

        #Print the contents of fields time and pfsupplies.voltage.value
        for i in range(len(my_itm_obj.pfsystemsArray.array)):
                print "CPO " + str(i) + ": time = " + str(my_itm_obj.pfsystemsArray.array[i].time) +
" profiles_1d.F_dia = " +  str(my_itm_obj.pfsystemsArray.array[i].pfsupplies.voltage.value)

read_cpo()
```

Python example to get a CPO slice

```
#Definition of the class structures in file ual.py
import sys
from ualdef import *
import ual
import numpy


'''
 This sample program reads the database filled by program put_cpo_slice.py. It takes a time argument
from the command line, and uses this value to get a time slice in the array of stored pfsystems
CPOs.
Linear interpolation is performed if the time lies between stored times. It the time lies outsize
the
range of stored times, the upper or lower CPO slice is considered.
'''

def read_cpo(time):

        #Program put_cpo_slice.py created database for shot 123, run 4
        #Here refShot and refRun arguments have no meaning
        my_itm_obj = ual.itm(123,4,123,0)
        my_itm_obj.open() #Open the database


        #Get the  CPO slice corresponding to the passed time
        #Defining linear interpolation. The other options are CLOSEST_SAMPLE and PREVIOUS_SAMPLE
        my_itm_obj.pfsystems.getSlice(time, INTERPOLATION)

        #Dump the whole returned CPO
        print my_itm_obj.pfsystems


if len(sys.argv) != 2:
        print "Usage: get_cpo_slice <time>"
else:
        read_cpo(float(sys.argv[1]))
```

3.4 JAVA Interface

The main class of the Java interface is **`UALAccess`** and is contained in package **`ualmemory.javainterface`**.
UALAccess defines a set of inner classes whose names correspond to the CPOs defined in the ITM simulation database.
Class `UALAccess` defines the following static methods:

- **`int open(String name, int shot, int run)`** to open a database instance. It returns the identifier to be used in the subsequent calls.
- **`int create(String name, int shot, int run, int refShot, int refRun)`** to create a new database instance (refShot and refRUn are currently not used). It returns the identifier to be used in the subsequent calls.
- **`void close(int refIdx)`** to close the current database
- **`void enableMemCaching(bool isShared, int cacheSize)`** to enable data caching in memory. Argument `isShared` specified whether the cache is shaed among processes or private. `cacheSize` specifies the dimension of the data cache.

When any method in the Java interface fails, an Exception is thrown. UAL methods must therefore be put into try blocks.
For every CPO type defined in the ITM simulation database, an inner class of UALAccess is defined, with the same name of the CPO type. The fields of the CPO inner class reflect the XML

definition of the corresponding CPO type. XML strings are translated to Java String instances, basic types are mapped onto the corresponding Java native type (boolean, int, float, double). Arrays are translated into support Vector classes. The Vector support classes defined in ualmemory.javainterface package are the following:

- `Vect1DBoolean`
- `Vect1DInt`
- `Vect1DFloat`
- `Vect1DDouble`
- `Vect1DString`
- `Vect2DInt`
- `Vect2DFloat`
- `Vect2DDouble`
- `Vect3DInt`
- `Vect3DFloat`
- `Vect3DDouble`
- `Vect4DInt`
- `Vect4DFloat`
- `Vect4DDouble`

Every Support Vect class has a constructor which accepts a native array, and defines the following accessor methods:

- **`int getDim()`** which returns the total number of elements for that array;
- **`int getElementAt(int idx, …)`** which returns the element at the specified index;
- **`void setElementAt(int idx,…, <type> element)`** which puts the passed element (whose type depends on the vector type) at the specified index(es);
- **`int[] getArray()`** which returns the set of elements as a 1D native array (using row-first ordering);

If the CPO is not time dependent, the corresponding class defines the following static methods:

- **`<CPO type> get(int expIdx, String path)`** to read the CPO instance from the database at the specified path. To use the default CPO instance, specify as path the name of the CPO type, otherwise specify <CPO type>/inst, where inst is the instance number for that CPO. Argument expIdx is the index returned by open() or create().
- **`void put(int expIdx, String path, <CPO class> cpo)`** to write the content of the passed CPO instance in the database;
- **`void flush(int expIdx, String path)`** to flush data cached in memory for the specified CPO.

If the CPO is time dependent, the corresponding class defines the following static methods:

- **`<CPO class> [] get(int expIdx, String path)`** which read the whole array of CPOs stored in the simulation database at the specified path. To use the default

CPO instance, specify as path the name of the CPO type, otherwise specify <CPO type>/inst, where inst is the instance number for that CPO. Argument expIdx is the index returned by open() or create();

– <CPO class> getSlice(int expIdx, String path, double time, int interpolMode) which reads a single CPO instance corresponding to the passed time;

– void put(int expIdx, String path, <CPO class> cpos[]) which writes the whole array of CPOs in the simulation database;

– void putSlice(int expIdx, String path, <CPO class> cpo) which appends the passed CPO instance to the current CPO array in the database. Note that this operation writes only time dependent CPO fields;

– void replaceLastSlice(int expIdx, String path, <CPO class> cpo) which replaces the last element of the CPO array in the database;

– void putNonTimed(int expIdx, String path, <CPO class> cpo) which writes in the database the fields of the CPO which do not depend on time;

– void flush(int expIdx, String path) to flush data cached in memory for the specified CPO.

Empty fields in Java CPOs are managed as follows :

- Empty integer fields are initialised as -999999999
- Empty real fields are initialised as -9.D40
- Empty vectors are initialised as « null ». It is responsibility of the programmer to test for the existence of vectors before using Vector accessor methods.

3.3.1 A JAVA example for reading and writing CPOs

```
package ualmemory.javainterface;

import java.io.*;

//Test class for the UAL Java interface
//The main class of the Java interface ia UALAccess. It defines a set of inner classes
//whose names correspond
//to the CPOs defined in the UAL ITM database.


class TestUAL
{
    static int expIdx;



//A sample method to write a some CPO arrays using method put
    static void samplePut()
    {

        //Create a new instance of the database
        try {
        expIdx = UALAccess.create("euitm", 123, 1, 123, 0);
        }catch(Exception exc) {System.err.println("Error creating UAL database: " +
exc);}

        //Instantiate an array of pfsystems objects
        UALAccess.pfsystems [] pfs = new UALAccess.pfsystems[10];
        for(int i = 0; i < 10; i++)
            pfs[i] = new  UALAccess.pfsystems();

        //Fill some fields
        for(int i = 0; i < 10; i++)
        {
            //Those fields are not time dependent. Even if in this example their
            //values are filled for all
            //the elements of the CPO array, only pfs[0] is considered in the
```

```
            //put() method.

            //A sample string
            pfs[i].datainfo.dataprovider = "SAMPLE PROVIDER";

            //A sample string array. A set of support vector classes is
            //defined in the UAL java interface
            //the naming convention for vector classes is vect<num dimensions>D<type>
            pfs[i].pfcoils.desc_pfcoils.name = new Vect1DString(2);
            pfs[i].pfcoils.desc_pfcoils.name.setElementAt(0, "SAMPLE 1");
            pfs[i].pfcoils.desc_pfcoils.name.setElementAt(1, "SAMPLE 2");

            //A 1 dimensional sample vector
            pfs[i].pfcoils.desc_pfcoils.res = new Vect1DDouble(100);
            for(int j = 0; j < 100; j++)
                pfs[i].pfcoils.desc_pfcoils.res.setElementAt(j, j*100);
            //A three dimensional sample vector
            pfs[i].pfcoils.desc_pfcoils.pfelement.pfgeometry.rzcoordinate.r =
               new Vect3DDouble(2,3,4);
            for(int j = 0; j < 2; j++)
            {
                for(int k = 0; k < 3; k++)
                {
                    for(int h = 0; h < 4; h++)
                    {
             pfs[i].pfcoils.desc_pfcoils.pfelement.pfgeometry.rzcoordinate.r.setElementAt
                    (j, k, h, 100*j+10*k+h);
                    }
                }
            }
           //The following field is time dependent. All the elements of the CPO
           //array are considered

            //A sample time dependent 1D vector
            pfs[i].pfsupplies.voltage.value = new Vect1DDouble(4);
            pfs[i].pfsupplies.voltage.value.setElementAt(0, i);
            pfs[i].pfsupplies.voltage.value.setElementAt(1, i+1);
            pfs[i].pfsupplies.voltage.value.setElementAt(2, i+2);
            pfs[i].pfsupplies.voltage.value.setElementAt(3, i+3);

            //Never forget to write the time for time dependent CPO instances!!!!
            pfs[i].time = i;
        }

        //The CPO array is now ready, let's write it into the database in a single
        //put() operation.
       //Since it may launch a UALException exception it is inserted in a try block;

        try {
            UALAccess.pfsystems.put(expIdx, "pfsystems", pfs);
        }catch(Exception exc) {System.err.println("Error writing in UAL: " + exc);}

        try {
            UALAccess.close(expIdx, "euitm", 123,1);
        }catch(Exception exc) {System.err.println("Error closing database: " + exc);}

    }

    /*************************************************************************8
    static void samplePutSlice()
    {
       //This method does the same as samplePut(), using method putSlice() instead
       //Create a new instance odf the database
        try {
        expIdx = UALAccess.create("euitm", 123, 1, 123, 0);
        }catch(Exception  exc)  {System.err.println("Error  creating  UAL  database:  "  +
exc);}

        //Now it suffice to instantiate a single CPO
        UALAccess.pfsystems pf = new UALAccess.pfsystems();

        //Save first those fields which are not time dependent

        //A sample string
        pf.datainfo.dataprovider = "SAMPLE PROVIDER";

        //A sample string array. A set of supporty vector classes is defined
       //in the UAL java interface
        //the naming convention for vector classes is vect<num dimensions>D<type>
```

```
        pf.pfcoils.desc_pfcoils.name = new Vect1DString(2);
        pf.pfcoils.desc_pfcoils.name.setElementAt(0, "SAMPLE 1");
        pf.pfcoils.desc_pfcoils.name.setElementAt(1, "SAMPLE 2");

        //A 1 dimensional sample vector
        pf.pfcoils.desc_pfcoils.res = new Vect1DDouble(100);
        for(int i = 0; i < 100; i++)
            pf.pfcoils.desc_pfcoils.res.setElementAt(i, i*10);
        //A three dimensional sample vector
        pf.pfcoils.desc_pfcoils.pfelement.pfgeometry.rzcoordinate.r =
              new Vect3DDouble(2,3,4);
        for(int i = 0; i < 2; i++)
            for(int j = 0; j < 3; j++)
                for(int k = 0; k < 4; k++)

             pf.pfcoils.desc_pfcoils.pfelement.pfgeometry.rzcoordinate.r.setElementAt
                   (i, j, k, 100*i+10*j+k);

        try {
            UALAccess.pfsystems.putNonTimed(expIdx, "pfsystems", pf);
        }catch(Exception exc) {System.err.println("Writing CPO slice: " + exc);}
        //Incrementally save time evolution for timed CPO fields

        //First allocate a Vect1DFloat object with dimension 4 for  field
       //pf.pfsupplies.voltage.value

        pf.pfsupplies.voltage.value = new Vect1DDouble(4);
        for(int i = 0; i < 2000; i++)
        {
            //Write array elements for sample i
            pf.pfsupplies.voltage.value.setElementAt(0, i);
            pf.pfsupplies.voltage.value.setElementAt(1, i+1);
            pf.pfsupplies.voltage.value.setElementAt(2, i+2);
            pf.pfsupplies.voltage.value.setElementAt(3, i+3);

            //Never forget the time
            pf.time = i;

            try {
                UALAccess.pfsystems.putSlice(expIdx, "pfsystems", pf);
            }catch(Exception  exc){System.err.println("Error  appending  CPO  slice:  "+
exc);}

        }

        try {
            UALAccess.close(expIdx, "euitm", 123,1);
        }catch(Exception exc) {System.err.println("Error closing database: " + exc);}

    }

//*****************************************************************************

    static void sampleGet()
    {
        UALAccess.pfsystems []retPfs;
        //Read the whole array of saved pfsystems CPOs
        try {
            expIdx = UALAccess.open("euitm", 123,1);
        }catch(Exception exc)
             {System.err.println("Error opening database: " + exc); return;}
        try {
            //retPfs = UALAccess.pfsystems.get(expIdx, "pfsystems");
            retPfs = UALAccess.pfsystems.get(expIdx, "pfsystems/1");
            //If get succesful, retPf contains the array of stopred pfsystems CPOs
       }catch(Exception exc){System.err.println("Error in get operation: " + exc);
             return;}

        System.out.println("Method get returned " + retPfs.length + " CPOs");
        for(int i = 0; i < retPfs.length; i++)
        {
            System.out.println("CPO "+ i + ":");
      //Write the contents of field pfsystems/pfsupplies/voltage/value
            for(int j = 0; j < retPfs[i].pfsupplies.voltage.value.getDim(); j++)
                System.out.print("" +
                    retPfs[i].pfsupplies.voltage.value.getElementAt(j));
            System.out.println("");
        }
         try {
```

```
            UALAccess.close(expIdx, "euitm", 123,1);
        }catch(Exception exc) {System.err.println("Error closing database: " + exc);}
    }

//*************************************************************************
    static void sampleGetSlice()
    {
        UALAccess.pfsystems retPf;
        //Read a resampled version of the stored CPO array
        try {
            expIdx = UALAccess.open("euitm", 123,1);
        }catch(Exception exc){System.err.println("Error opening database: " + exc);}

        //Get the CPO slice corresponding to time 5.6, performing linear interpolation
        try {
        retPf = UALAccess.pfsystems.getSlice(expIdx, "pfsystems",
              5.6, UALAccess.INTERPOLATION);
        }catch(Exception exc){System.err.println("Error getting CPOslice: " + exc);
             return;}

        //Utility method dump is defined for all CPOs and writes the content of the CPO
        //instance.
        retPf.dump();
         try {
            UALAccess.close(expIdx, "euitm", 123,1);
        }catch(Exception exc) {System.err.println("Error closing database: " + exc);}
    }
```

4 The Low Level Layer API

All the high level UAL interfaces never perform directly I/O in the underlying database, but use the API provided by a low level library. Currently, the low level library provides support for MDSplus databases and HDF5 files (currently under test).
This section is of interest only for developers of new high level UAL interfaces, since none of the exported routines is visible in the high level interface.
The interface listed below is derived by the include file `ual_low_level.h`, which defines all the prototypes of the low level library. No data structures are defined at this level, and get/put operation refer always to a single data item, specified by the combination of arguments `cpoPath` and `path`. `cpoPath` is the path name of the CPO root in the ITM simulation database (for the default instance this corresponds to the CPO type in the database XML definition, otherwise it is <CPO type>/<instance number>. `path` is the path name (expressed using XPath Syntax) of the CPO fields with reference to the CPO root. All the get and put routines define as first argument the integer identifier returned by `euitm_open()` or `euitm_create()`.
For time-independent CPOs there is a straighfoward mapping between the types of the CPO fields and the data types stored in the database. The same holds for time-independent fields of time-dependent CPOs. For time-dependent fieds, the dimension is increased by one (with respect to the dimension declared in the XML definition) in order to account for the fact that the database field is storing data for a (unidimensional) array of CPO samples. For example, if a CPO time-dependent field is declared as scalar in the XML definition (and therefore will be represented by a scalar value in the high level representation of that CPO), it will be stored as an 1D array in the simulation database, where each sample corresponds to the value of the field in the corresponding CPO sample.

The mapping between N-dimensional database fields and N-1-dimensional arrays in the corresponding CPO field for a time-dependent CPO array, as well as packing single fields into CPO objects, is performed by the high level layer.
For every I/O operation handling arrays, a pointer to the corresponding linearized data array is passed. The low level UAL assumes **column-first** ordering for multidimensional arrays, where the last dimension is the dimension of the CPO array (for stored arrays describing time-dependent CPO fields). For example a time-dependent CPO field which is represented by a 1D array with 10 elements will be stored in the simulation database, for an array of 5 CPO samples, as a (10x5) bidimensional array.

All the get/put reoutines return an integer status. 0 means success, otherwise, the corresponding error message is retrieved via routine `*euitm_last_errmsg()`

4.1 put routines

The put routines use the following naming convention:
`put<type>()` for writing scalar values and **`putVect<dim>D<type>()`** for writing arrays (in this case, besides the pointer to the data array, the array dimension(s) are also passed as well as a Boolean flag which indicates whether data refer to a time-dependent CPO field.

```
int putString(int expIdx, char *cpoPath, char *path, char *data);
int putInt(int expIdx, char *cpoPath, char *path, int data);
int putFloat(int expIdx, char *cpoPath, char *path, float data);
int putDouble(int expIdx, char *cpoPath, char *path, double data);
int putVect1DString(int expIdx, char *cpoPath, char *path, char **data, int dim,
int isTimed);
int putVect1DInt(int expIdx, char *cpoPath, char *path, int *data, int dim, int
isTimed);
int putVect1DFloat(int expIdx, char *cpoPath, char *path, float *data, int dim,
int isTimed);
int putVect1DDouble(int expIdx, char *cpoPath, char *path, double *data, int
dim, int isTimed);
int putVect2DInt(int expIdx, char *cpoPath, char *path, int *data, int dim1, int
dim2, int isTimed);
int putVect2DFloat(int expIdx, char *cpoPath, char *path, float *data, int dim1,
int dim2, int isTimed);
int putVect2DDouble(int expIdx, char *cpoPath, char *path, double *data, int
dim1, int dim2, int isTimed);
int putVect3DInt(int expIdx, char *cpoPath, char *path, int *data, int dim1, int
dim2, int dim3, int isTimed);
int putVect3DFloat(int expIdx, char *cpoPath, char *path, float *data, int dim1,
int dim2, int dim3, int isTimed);
int putVect3DDouble(int expIdx, char *cpoPath, char *path, double *data, int
dim1, int dim2, int dim3, int isTimed);
int putVect4DInt(int expIdx, char *cpoPath, char *path, int *data, int dim1, int
dim2, int dim3, int dim4, int isTimed);
int putVect4DFloat(int expIdx, char *cpoPath, char *path, float *data, int dim1,
int dim2, int dim3, int dim4, int isTimed);
int putVect4DDouble(int expIdx, char *cpoPath, char *path, double *data, int
dim1, int dim2, int dim3, int dim4, int isTimed);
```

4.2 putSlice routines

The putSlice routines use the following naming convention:
`put<type>Slice()` for writing scalar values and **`putVect<dim>D<type>Slice()`** for writing arrays. Note that in this case the dimension of the CPO fields is the same of that declared in the XML description, even for time-dependent fields. putSlice routine will also require the time argument, i.e. the time to which the CPO sample refers to.

```
int putIntSlice(int expIdx, char *cpoPath, char *path, int data, double time);
int putFloatSlice(int expIdx, char *cpoPath, char *path, float data, double time);
int putDoubleSlice(int expIdx, char *cpoPath, char *path, double data, double time);
int putStringSlice(int expIdx, char *cpoPath, char *path, char *data, double time);
int putVect1DIntSlice(int expIdx, char *cpoPath, char *path, int *data, int dim, double time);
int putVect1DFloatSlice(int expIdx, char *cpoPath, char *path, float *data, int dim, double time);
int putVect1DDoubleSlice(int expIdx, char *cpoPath, char *path, double *data, int dim, double time);
int putVect2DIntSlice(int expIdx, char *cpoPath, char *path, int *data, int dim1, int dim2, double time);
int putVect2DFloatSlice(int expIdx, char *cpoPath, char *path, float *data, int dim1, int dim2, double time);
int putVect2DDoubleSlice(int expIdx, char *cpoPath, char *path, double *data, int dim1, int dim2, double time);
int putVect3DIntSlice(int expIdx, char *cpoPath, char *path, int *data, int dim1, int dim2, int dim3, double time);
int putVect3DFloatSlice(int expIdx, char *cpoPath, char *path, float *data, int dim1, int dim2, int dim3, double time);
int putVect3DDoubleSlice(int expIdx, char *cpoPath, char *path, double *data, int dim1, int dim2, int dim3, double time);
int putVect4DIntSlice(int expIdx, char *cpoPath, char *path, int *data, int dim1, int dim2, int dim3, int dim4, double time);
int putVect4DFloatSlice(int expIdx, char *cpoPath, char *path, float *data, int dim1, int dim2, int dim3, int dim4, double time);
int putVect4DDoubleSlice(int expIdx, char *cpoPath, char *path, double *data, int dim1, int dim2, int dim3, int dim4, double time);
```

4.3 get routines

The get routines use the following naming convention:
`get<type>()` for reading scalar values and **`getVect<dim>D<type>()`** for reading arrays.
Besides the common `expIdx`, `cpoPath` and `path` arguments, these routines define a pointer to the data buffer pointer. The data buffer is allocated by the get routine (this is why an additional level of pointing is required), and has to be freed by the program after its usage via the stdlib `free()` routine. The dimension(s) of the array is returned in getVect routines.

```
int getString(int expIdx, char *cpoPath, char *path, char **data);
int getFloat(int expIdx, char *cpoPath, char *path, float *data);
int getInt(int expIdx, char *cpoPath, char *path, int *data);
int getDouble(int expIdx, char *cpoPath, char *path, double *data);
int getVect1DString(int expIdx, char *cpoPath, char *path, char ***data, int *dim);
int getVect1DInt(int expIdx, char *cpoPath, char *path, int **data, int *dim);
int getVect1DFloat(int expIdx, char *cpoPath, char *path, float **data, int *dim);
int getVect1DDouble(int expIdx, char *cpoPath, char *path, double **data, int *dim);
int getVect2DInt(int expIdx, char *cpoPath, char *path, int **data, int *dim1, int *dim2);
int getVect2DFloat(int expIdx, char *cpoPath, char *path, float **data, int *dim1, int *dim2);
int getVect2DDouble(int expIdx, char *cpoPath, char *path, double **data, int *dim1, int *dim2);
int getVect3DInt(int expIdx, char *cpoPath, char *path, int **data, int *dim1, int *dim2, int *dim3);
int getVect3DFloat(int expIdx, char *cpoPath, char *path, float **data, int *dim1, int *dim2, int *dim3);
int getVect3DDouble(int expIdx, char *cpoPath, char *path, double **data, int *dim1, int *dim2, int *dim3);
int getVect4DInt(int expIdx, char *cpoPath, char *path, int **data, int *dim1, int *dim2, int *dim3, int *dim4);
int getVect4DFloat(int expIdx, char *cpoPath, char *path, float **data, int *dim1, int *dim2, int *dim3, int *dim4);
```

```
int getVect4DDouble(int expIdx, char *cpoPath, char *path, double **data, int
*dim1, int *dim2, int *dim3, int *dim4);
```

4.4 getSlice routines

The getSlice routines use the following naming convention:
`get<type>Slice()` for reading scalar values and **`getVect<dim>D<type>Slice()`** for reading arrays. They are valid only for time-dependent CPO fields, and the returned value has the same dimensionality of what is declared in the XML definition (although time-dependent firlds are stored in the simulation database with an additional dimension).
Besides the common `expIdx`, `cpoPath` and `path` arguments, these routines define a pointer to the data buffer pointer. The data buffer is allocated by the get routine (this is why an additional level of pointing is required), and has to be freed by the program after its usage via the stdlib `free()` routine. The dimension(s) of the array is returned in getVect routines. Three more arguments are passed with respect to the get() routines:

- `double time`: the time corresponding to the data slice to be retrieved;
- `int interpolMode`: the kind of interpolation, i.e either INTERPOLATION, CLOSEST_SAMPLE, PREVIOUS_SAMPLE
- `double *retTime`: the returned value of the time. This corresponds to the passed time when INTERPOLATION is selected, otherwise it is the time associated to the selected sample.

```
int getStringSlice(int expIdx, char *cpoPath, char *path, char **data, double
time, double *retTime, int interpolMode);
int getFloatSlice(int expIdx, char *cpoPath, char *path, float *data, double
time, double *retTime, int interpolMode);
int getIntSlice(int expIdx, char *cpoPath, char *path, int *data, double time,
double *retTime, int interpolMode);
int getStringSlice(int expIdx, char *cpoPath, char *path, char **data, double
time, double *retTime, int interpolMode);
int getDoubleSlice(int expIdx, char *cpoPath, char *path, double *data, double
time, double *retTime, int interpolMode);
int getVect1DIntSlice(int expIdx, char *cpoPath, char *path, int **data, int
*dim, double time, double *retTime, int interpolMode);
int getVect1DFloatSlice(int expIdx, char *cpoPath, char *path, float **data, int
*dim, double time, double *retTime, int interpolMode);
int getVect1DDoubleSlice(int expIdx, char *cpoPath, char *path, double **data,
int *dim, double time, double *retTime, int interpolMode);
int getVect2DIntSlice(int expIdx, char *cpoPath, char *path, int **data, int
*dim1, int *dim2, double time, double *retTime, int interpolMode);
int getVect2DFloatSlice(int expIdx, char *cpoPath, char *path, float **data, int
*dim1, int *dim2, double time, double *retTime, int interpolMode);
int getVect2DDoubleSlice(int expIdx, char *cpoPath, char *path, double **data,
int *dim1, int *dim2, double time, double *retTime, int interpolMode);
int getVect3DIntSlice(int expIdx, char *cpoPath, char *path, int **data, int
*dim1, int *dim2, int *dim3, double time, double *retTime, int interpolMode);
int getVect3DFloatSlice(int expIdx, char *cpoPath, char *path, float **data, int
*dim1, int *dim2, int *dim3, double time, double *retTime, int interpolMode);
int getVect3DDoubleSlice(int expIdx, char *cpoPath, char *path, double **data,
int *dim1, int *dim2, int *dim3, double time, double *retTime, int
interpolMode);
```

4.5 replaceLastSlice routines

The replaceLastSlice routines use the following naming convention:
`replaceLast<type>Slice()` for replacing the last scalar slice in the database and **`replaceLastVect<dim>D<type>Slice()`** for replacing the last array slice. Observe that only the last slice stored in the database can be replaced. If more slices have to be replaces, it is necessary to delete the whole field (using deleteData()) and to store again all the data slices.

```
int replaceLastIntSlice(int expIdx, char *cpoPath, char *path, int data);
```

int replaceLastFloatSlice(int expIdx, char *cpoPath, char *path, float data);
int replaceLastDoubleSlice(int expIdx, char *cpoPath, char *path, double data);
int replaceLastStringSlice(int expIdx, char *cpoPath, char *path, char *data);
int replaceLastVect1DIntSlice(int expIdx, char *cpoPath, char *path, int *data, int dim);
int replaceLastVect1DFloatSlice(int expIdx, char *cpoPath, char *path, float *data, int dim);
int replaceLastVect1DDoubleSlice(int expIdx, char *cpoPath, char *path, double *data, int dim);
int replaceLastVect2DIntSlice(int expIdx, char *cpoPath, char *path, int *data, int dim1, int dim2);
int replaceLastVect2DFloatSlice(int expIdx, char *cpoPath, char *path, float *data, int dim1, int dim2);
int replaceLastVect2DDoubleSlice(int expIdx, char *cpoPath, char *path, double *data, int dim1, int dim2);
int replaceLastVect3DIntSlice(int expIdx, char *cpoPath, char *path, int *data, int dim1, int dim2, int dim3);
int replaceLastVect3DFloatSlice(int expIdx, char *cpoPath, char *path, float *data, int dim1, int dim2, int dim3);
int replaceLastVect3DDoubleSlice(int expIdx, char *cpoPath, char *path, double *data, int dim1, int dim2, int dim3);
int replaceLastVect4DIntSlice(int expIdx, char *cpoPath, char *path, int *data, int dim1, int dim2, int dim3, int dim4);
int replaceLastVect4DFloatSlice(int expIdx, char *cpoPath, char *path, float *data, int dim1, int dim2, int dim3, int dim4);
int replaceLastVect4DDoubleSlice(int expIdx, char *cpoPath, char *path, double *data, int dim1, int dim2, int dim3, int dim4);

4.6 begin/end routines

This set of routines has to be called by the high level layer when either a get(), put(), getSlice(), putSlice() sequence is going to begin. When applied to a CPO (array) these operations are translated by the high level layer into a sequence of similar operations for every field of the current CPO. For example, in order to put a CPO into the simulation database, a sequence of put operations will be performes, one for every field of the target CPO. In this case it will be required that beginCPOPut() be called before the sequence of put() operations, and endCPOPut() be called afterwards. The routines below are possibly used by the low level layer to properly configure the underlying system and prepare for the following operations. For these routines, the path argument refers to the path name of the CPO root.

```
int beginCPOGet(int expIdx, char *path, int isTimed, int *retSamples);
void endCPOGet(int expIdx, char *path);
```

to be called before and after a sequence of get operations for a given CPO. beginCPOGet returns also the number of stored CPO samples.

```
int beginCPOGetSlice(int expIdx, char *path, double time);
void endCPOGetSlice(int expIdx, char *path);
```

to be called before and after a sequence of getSlice operations for a given CPO. beginCPOGetSlice requires the time which will be used by the following getSlice() calls.

```
int beginCPOPut(int expIdx, char *path);
void endCPOPut(int expIdx, char *path);
```

to be called before and after a sequence of put() calls for time-independent CPOs

```
int beginCPOPutTimed(int expIdx, char *path, int samples, double *inTimes);
int endCPOPutTimed(int expIdx, char *path);
```

to be called before and afte ra sequence of put() calls for time-dependent fields of time-dependent CPOs

```
int beginCPOPutNonTimed(int expIdx, char *path);
void endCPOPutNonTimed(int expIdx, char *path);
```

to be called before and after a sequence of put() calls for time-indedependent fields of time-dependent CPOs

```
int beginCPOPutSlice(int expIdx, char *path);
void endCPOPutSlice(int expIdx, char *path);
```

to be called before and afte ra sequence of putSlice() calls for a given CPO

```
int beginCPOReplaceLastSlice(int expIdx, char *path);
void endCPOReplaceLastSlice(int expIdx, char *path);
```

to be called before and afte ra sequence of replaceLastSlice() calls for a given CPO

4.7 Miscellanea

```
int deleteData(int expIdx, char *cpoPath, char *path);
```

delete all data for that CPO field (for all CPOs in in a CPO array)

```
char *euitm_last_errmsg();
```

return the last error message. To be called when the returned status is not 0.

```
int euitm_create(char *name, int shot, int run, int refShot, int refRun, int
*retIdx);
```

create a new database, corresponding to shot and run number. The name argument must be "euitm". The returned retIdx argument will be used in the subsequent get/put routines.

```
int euitm_open(char *name, int shot, int run, int *retIdx);
```

open an existing database, corresponding to shot and run number. The name argument must be "euitm". The returned retIdx argument will be used in the subsequent get/put routines.

```
int euitm_close(int idx);
```

close the open database.

```
void euitm_enable_mem_cache(int isShared, int size)
```

enable data caching in memory. If argument isShared is true, the memory is shared among processes. Argument size specifies the dimension of the cache.

```
void euitm_disable_mem_cache();
```

disable data caching.

```
void euitm_discard_mem(int expIdx, char *cpoPath, char *path);
```

reclaim cache memory used for that data.

```
void euitm_flush(int expIdx, char *cpoPath, char *path)
```

flushes cached data for that CPO field