Physics Data Model and Access Layer User Guide

Version June 2021

(applies to Access Layer 4.9.0 and beyond)

Contents

1 Preamble: key principles of the Physics Data Model

The Physics Data Model provides information for data providers and data consumers on:

- What data exist ?
- What are they called ?
- How are they structured as seen by the user ?

The Data Model therefore provides a unique definition for all users of Physics data, separating the « what is there ? » (data model) from the « where do I find it ? » (storage methods) and the « how do I get it ? » (access methods).

The Data Model is used to describe ITER experimental and simulated data, as well as being able to describe data from any other fusion experiment. The data structures are identical for experimental and simulated data, which facilitates the simulation/experiment comparison.

The Data Model is extensible and can evolve through Lifecycle procedures described in a separate document.

A powerful feature of the Data Model is the automated definition of the data structures for all supported languages.

The Data Model has a tree structure and is intended to be accessible at any level of the tree, from a single quantity (leaf node) to higher level structures. Within the Data Model, some structures are marked as Interface Data Structure (IDS), a very important notion. An IDS is an entry point of the Data Model that can be used as a single entity to be used by a user. Examples are the full description of a tokamak subsystem (diagnostic, heating system, …) or an abstract physical concept (equilibrium, set of core plasma profiles, wave propagation, …). This concept allows tracing of data provenance and allows a simple transfer of large numbers of variables between loosely or tightly coupled applications. The IDS thereby define standardized interface points between IMAS physics components. An IDS is a part of the Data Model, an entry point into it, thus the IMAS components are interfaced with the same structures as those constituting the Data model. An IDS is marked by having a child `ids_properties` node, containing traceability and self-description information.

Within the context of a tree structure, we define the following: a node is any element of a tree, a leaf is a node which is an end-point of a tree, a parent is one level above a particular node, a sibling is a node sharing the same parent with a particular node and a child is one level below a particular node. Nodes have either children or data, not both.

A Data Entry is a collection of IDSs (potentially all IDSs) and their multiple occurrences (see 4.1). It is a database concept that allows grouping and storing data (compliant to the data model) as a single dataset with a unique reference. This concept is used whatever the location of the Data Entry, whether it is a temporary data entry used only for communication within an integrated modelling workflow (resides in memory or in local files) or whether it is a dataset to be stored in a local or remote database.

The data model defines structures that are usable by the IMAS supported programming languages. Sections 2-4 explain how these structures are defined, their properties and their usage. At some point, the Access Layer may be used to read or write these structures to a storage, which may be local, remote, or in memory. The Access Layer is documented in section 5.

2 Node properties

This section describes the various properties of the nodes of the Data Model. These properties appear in the Data Model documentation (type `dd_doc` in the command line under the IMAS environment to display the data model documentation).

2.1 Path

The path of a node (i.e. its position in the Data Model tree structure) is indicated in the "Full path name" column of the documentation, relative to the nearest parent IDS.

2.2 Node and data types, default values

The table below lists the available node/data types, as well as how they are implemented in the supported programming languages. For data nodes, the data type implementation is not filled for Matlab and Python since they are interpreted languages. The rightmost column of the table indicates the default value returned by the access methods when a node is empty.

Arrays of structures that are parents of nodes with different timebases have a maximum size indicated in the documentation (e.g. [max_size=20]). This is a limitation of the present implementation, to be removed in the future.

The present Fortran implementation uses the F90 standard but it is planned to upgrade soon to the F2003 standard in order to benefit from many improvements (e.g. allocatable character strings).

The reason why default values for scalar integers and reals is a reserved value instead of NaN is that NaN are not handled in the same way by the various C++ and Fortran compilers and would create compiler-dependent behavior of the physics components.

Node/Data type	***Definition***	***Fortran implementation***	***C++ implementation***	***Java implementation***	***Matlab implementation***	***Python implementation***	***Default value***
Structure	Structure (node	Type	Class	Class	Structure	Class	

	with children)	All type names have the prefix `ids_` Example: `type(ids_equilibrium) :: equilibrium`					
struct_array	Array of Structure	Type, pointer (:) Example: `type (ids_magnetics_flux_loop),pointer :: flux_loop(:) => null()`	Array of class	Arrays of class	Cell array	Class containing array of classes	Not allocated (declarative languages) or of zero size (Matlab, Python)
INT_0D, INT_1D, INT_2D, INT_3D	Integer and arrays of integers	Integer, pointer (for arrays) Example (INT_0D): `integer(ids_int) :: turns=-999999999` The `ids_is_valid(node)` function allows testing whether the node contains a non-default value	int array of int	int, Vect1DInt, Vect2DInt, Vect3Dint	int array of int	int NumPy.ndarray[int, ndim=n] (n=1..3)	Scalar: -999999999 Arrays: not allocated (declarative languages) or of zero size (Matlab, Python)

FLT_0D, FLT_1D, FLT_2D, FLT_3D, FLT_4D, FLT_5D, FLT_6D	Real and arrays of reals	Real, double precision, pointer (for arrays) Example (FLT_1D): `real(ids_real), pointer :: data(:) => null()` The `ids_is_valid(node)` function allows testing whether the node contains a non-default value	double Array of double	double, Vect1DDouble, Vect2DDouble, Vect3DDouble, Vect4DDouble, Vect5DDouble, Vect6DDouble	double Array of double	float NumPy.ndarray[float, ndim=n] (n=1..6)	Scalar: -9.0E40 Arrays: not allocated (declarative languages) or of zero size (Matlab, Python)
STR_0D, STR_1D	String and arrays of strings	character(len=132), dimension(:), pointer for both cases. The STR_0D type is implemented as a vector of strings of length 132 characters, allowing arbitrary long strings. STR1D implementation is for the moment limited to a vector of 132 character strings. Example: `character(len=13`	char array of char	String, Vect1DString	char array of char	Str List of strings	Not allocated (declarative languages) or of zero size (Matlab, Python)

		2), dimension(:), pointer ::name => null()					
CPX_0D, CPX_1D, CPX_2D, CPX_3D, CPX_4D, CPX_5D, CPX_6D	Complex number and arrays of complex numbers.	Complex, double precision, pointer (for arrays) Example (CPX_1D): complex(ids_real), pointer :: data(:) => null()	std_complex_t defined in UALDef.h: typedef std::complex < double > std_complex_t;	Complex Vect1DComplex Vect2DComplex Vect3DComplex Vect4DComplex Vect5DComplex Vect6DComplex	Double (with complex attribute) Array of double (with complex attribute)	complex NumPy.ndarray[np.complex128, ndim=n] (n=1..6)	Scalar: (-9.0E40, -9.0E40i) Arrays: not allocated (declarative languages) or of zero size (Matlab, Python)

2.3 Time-variation

The Data Model distinguishes between categories of data according to their time-variation; "constant" data are data which are not varying within the context of the data being referred to (e.g. pulse, simulation, calculation); "static" data are likely to be constant over a wider range (e.g. nominal coil positions during operation); "dynamic" data are those which vary in time within the context of the data.

Dynamic data have a “time” coordinate, with two possible locations depending on the ids_properties/homogeneous_time value, see 4.3.

This time-variation property appears in brackets {} in the Description column of the documentation.

2.4 Units

Float and Complex data items have their units defined. If the quantity is dimensionless, the units shall be “-“; if the quantity has use case dependent units, e.g. the proportional gain of a linear controller `linear_controller/pid/p`, then the units shall be “mixed”.

This property appears in brackets [] in the Description column of the documentation.

2.5 Coordinates

The N coordinates of an N-dimensional array are documented in the Data Model.

When a coordinate is another quantity described in the Data Model, its path (relative to the nearest parent IDS) is indicated in the “Coordinates” column of the documentation.

When a coordinate is just a set of indices, it is indicated as “1…N” in the “Coordinates” column of the documentation, with no significance of the value N other than being the number of items in this specific node. Some nodes have a fixed number of items, in that case the value of N is indicated (e.g. “1…3”).

Time coordinates have a special usage rule in the context of an IDS, see 4.3.

3 Set of standard structures

Standard structures are data structures that can be found in various places of the Physics Data Model as they are useful in several contexts. Typical examples are lists of standard spatial coordinates, descriptions of plasma ion species, traceability / provenance information, etc.

The list of available reusable structures can be found in the data model documentation. This list is not restrictive since it can be freely expanded by Data Model designers. Note that the structure name is not the name of a Data Model node, therefore the generic structure names do not appear in the Data Model HTML documentation. They are primarily used for the design of the Data Dictionary, but can also be used in Fortran codes where they are implemented as derived types.

4 Interface Data Structures (IDS)

An interface Data Structure is marked by the presence of a child `ids_properties` node. It also has a `time` and a `code` child.

The list of existing IDS is provided in the top table of the HTML documentation.

4.1 IDS occurrences

There can be multiple instances, or "occurrences" of a given IDS in a Database Entry (see 5.2) or used in an IMAS workflow. These occurrences can correspond to different methods for computing the physical quantities of the IDS, or to different functionalities in a workflow (e.g. store initial values, prescribed values, values at next time step, …), or even to multiple subsystems (e.g. diagnostics) of the same type in an experiment.

By default, the IDS name without specification of the occurrence number (e.g. "equilibrium") corresponds to occurrence "0". IDS occurrences above the default value (occurrence "0") are accessed by concatenating the name of the IDS with the occurrence number, with a "/" in between. For example "equilibrium/2" is the name of the occurrence number 2 of the equilibrium IDS. Note that "equilibrium/0" is not valid (temporary limitation).

In the implementation of the MDS+ backend only, there is a pre-set maximum number of occurrences of a given IDS usable in a Database entry or in a workflow. This number is indicated in the documentation in the "Max. occurrence number" column of the list of IDS table. This limitation doesn't apply to other backends.

4.2 Empty fields

An IDS can contain a fairly large number of physical quantities and covers a wide range of applications. Therefore there will be many cases in which they are only partially filled. This is of course handled by the access software.

The only requirement regarding empty fields are:

- The ids_properties/homogeneous_time field must be filled
- When a quantity is filled, the coordinates of this quantity must be filled as well

Not meeting these requirements when one of the coordinates is a time will cause PUT methods (see 5.9) to return an error.

Empty fields are implemented in IDS variables as described in 2.2.

4.3 Timebases

"time" is a reserved node name for any timebase in the Data Model.

An IDS may contain quantities with different timebases in order to have the ability to describe experimental data as it is acquired in the experiment. However, an IDS can also be filled in a synchronous way (i.e. all dynamic quantities are stored on a unique timebase) and declared so by setting the ids_properties/homogeneous_time value to 1, since this will likely be a frequent usage in IMAS workflows. There are therefore two possible usages of the IDS, with two possible locations for the "time" coordinate related to a given node.

Value of ids_properties/homogeneous_time	***Location of the time coordinate for dynamic nodes***
0	Dynamic nodes may be asynchronous, their timebase is located as indicated in the "Coordinates" column of the documentation.
1	All dynamic nodes are synchronous, their common timebase is the "time" node that is the child of the nearest parent IDS.
2	Means that no dynamic node is filled in the IDS (dynamic nodes will skipped by the Access Layer)

4.4 IDS variables in programming languages

The IMAS infrastructure provides libraries for defining IDSs so that they can be directly used as variables in the supported programming languages. The implementation of the various data types is given in section 2.2. They are then manipulated as standard variables of the programming language.

Some specific functions are provided to help manipulate IDS variables, as listed below.

4.4.1 Initializing an empty IDS

IDS variables are initialized with default values (see 2.2).

For Fortran and C++, the declaration of the IDS variables is sufficient to initialize them.

In Java, empty IDS variables are initialized by:
`imas.<IDSname> empty_ids_variable = new imas.<IDSname>();`
where `<IDSname>` is the name of the IDS that the new variable has to represent.

In Python, classes representing IDSs are defined as member variables of class 'imas'. They are initialized as : `empty_ids_variable = imas.<IDSname>()`. Example : `empty_ids_variable = imas.pf_active()`

In Matlab, empty IDS variables are initialized by using the `ids_init` function:
Syntax : `empty_ids_variable = ids_init(IDSname)`

Example : `eq = ids_init('equilibrium')`
The array of structures in the resulting IDS will be empty (size=0). Use `ids_allocate` to allocate each array of structures to the desired size.
Syntax : `empty_ids_variable = ids_allocate(IDSname, AOSpath, size)`
Example : `eq.time_slice = ids_allocate('equilibrium','time_slice',5)`
This 2-steps allocation method deprecates the older initialization method `ids_gen`, which is still present for backward compatibility but should be avoided now. This method was allocating all array of structure of the created empty_ids_variable to size 1, which created issues (unused array of structures were also set to size 0 while they should have been of size 0 (i.e. set it to {}); forgetting to pre-allocate an AoS to its full size when it has more than one element) – see IMAS-3112 on JIRA.

4.4.2 Copying an IDS variable

The following functions are available to do a deep copy of an IDS variable into another one. Using the "=" instead of these functions may result into an incomplete copy for some languages (Fortran, Python) and is thus not recommended.

Language	***Syntax of the copy function***
Fortran	`call ids_copy(ids_variable_in, ids_variable_out)` NB: the deep copy mechanism will be provided by the "=" assignment as soon as the IMAS Infrastructure moves to Fortran 2003 NB2: this function also allows copying any sub-structure of the data dictionary from an IDS variable to another. This is implemented in Fortran only for the moment. If the target sub-structure is an array of structure, it must be first allocated, then copied index by index.
C++	`ids_variable_out = ids_variable_in;`
Java	Not implemented yet
Python	`ids_variable_out = ids_variable_in.deepcopy()`
Matlab	`ids_variable_out = ids_variable_in`

4.4.3 Deallocating an IDS

The following functions are available to deallocate all fields of an IDS variable.

Language	***Syntax of the deallocate function***
Fortran	`call ids_deallocate(ids_variable)`
C++	`ids_variable.delete[];`
Java	Done by the garbage collector of the JVM
Python	Done by built-in mechanisms for memory management. Could be also done explicit by calling "`del var`"
Matlab	`ids_variable = []`

Note for Fortran developers: IDS variables are allocated in C (using compatible types) when obtained from GET or GET_SLICE calls of the Access-layer, and should be deallocated through a call to ids_deallocate. IDS variables directly allocated in Fortran can also be deallocated through ids_deallocate. But one should not mix Fortran allocations on aan IDS variable returned by a GET or GET_SLICE from the Access-layer, this will cause the ids_deallocate to fail. Deallocating a node or sub-structure of an IDS

The Fortran API allows deallocating a node or a sub-structures of an IDS.***Language***	***Syntax of the deallocate function***
Fortran	`call ids_deallocate_struct(ids_node, c_data)`

`ids_node` is the targeted sub-structure of the IDS in Fortran (derived type).

`c_data` is a logical, it should be `.true.` if the data of the IDS (or specifically of the targeted sub-structure) was obtained through the Access Layer, and `.false.` if it was allocated directly in Fortran.

4.5 Identifier structure

4.5.1 Definition

The "identifier" structure is a standard structure used in several places of the Data Dictionary. It is used to provide an enumerated list of options for defining e.g.:

- a particular coordinate system may be one of the following options: Cartesian, cylindrical, or spherical.
- a particle may be either an electron, an ion, a neutral atom, a molecule, a neutron, or a photon.

- plasma heating may come from neutral beam injection, electron cyclotron heating, ion cyclotron heating, lower hybrid heating, alpha particles.

Identifiers are a prescribed list of possible labels, thus a user is not allowed to invent a new label. Each label has three representations (the children of the identifier structure): an index (integer), a name (short string) and a description (long string). Examples (from part of the core_sources/source identifier):

- {index=2, name="NBI", description="Source from Neutral Beam Injection"}
- {index=3, name="EC", description="Sources from heating at the electron cyclotron heating and current drive"}
- {index=4, name="LH", description="Sources from lower hybrid heating and current drive"}
- {index=5, name="IC", description="Sources from heating at the ion cyclotron range of frequencies"}
- {index=6, name="fusion", description="Sources from fusion reactions, e.g. alpha particle heating"}

The list of possible labels for a given identifier structure in the Data Dictionary can be found in the Data Dictionary documentation "dd_doc" (execute the command "dd_doc" from the command prompt).

4.5.2 Identifiers library

The three representations of a label should always be consistently filled. To help the user automate this process a library has been developed. This library is implemented in Fortran, C++ and Python. The IMAS-identifier library can be accessed by:

`module load IMAS` (i.e. the same command as for accessing the data dictionary and the access layer)

```
pkg-config –flags –libs imas-identifier-<FortranCompilerName>
```

In the Fortran code: `use imas-<IdentifierName>, only: <IdentifierName>`

The derived type `<IdentifierName>-identifier` includes both a list of all identifier indices and a set of functions to generate the full name and the description from a given index.

Example 1 (Fortran): How to use the IMAS-identifier library to fill the NBI label in the core_sources IDS.

```
  use ids_schemas, only: ids_core_sources
  use imas_core_source_identifier, only: core_source_identifier
  type(ids_core_sources) :: core_sources
  allocate(core_sources%source(1))
  core_sources%source(1)%identifier%index = core_source_identifier%NBI
  core_sources%source(1)%identifier%name =
core_source_identifier%name( core_source_identifier%NBI )
  core_sources%source(1)%identifier%description =
core_source_identifier%description( core_source_identifier%NBI )
```

Example 2 (Fortran): How to use the IMAS-identifier library to fill the type of coordinate system used in the equilibrium IDS.

```
  use ids_schemas, only: ids_equilibrium
  use imas_poloidal_plane_coordinates_identifier, only:
IDENT=>poloidal_plane_coordinates_identifier
  type(ids_equilibrium) :: equilibrium
  allocate( equilibrium%time_slice(1) )
  allocate( equilibrium%time_slice(1)%profiles_2d(1) )
  equilibrium%time_slice(1)%profiles_2d(1)%grid_type%index =
IDENT%rectangular
  equilibrium%time_slice(1)%profiles_2d(1)%grid_type%name =
IDENT%name( IDENT%rectangular )
  equilibrium%time_slice(1)%profiles_2d(1)%grid_type%description =
IDENT%description( IDENT%rectangular )
```

5 Access Layer

This section describes how to use the Access Layer to GET and PUT data compliant with the Physics Data Model.

The present implementation of the Access Layer only contains methods for GETTING and PUTTING IDSs. Later, similar methods will be provided to access data with a finer granularity (down to single leaf access).

By default, data are stored locally as MDS+ files, but other backends will be presented in section 5.14.

5.1 Getting started

On the ITER cluster, IMAS is configured using environment variables. To use the latest version (default IMAS module), type:

```
$ module load IMAS
```

To load a specific IMAS version, type the specific module name (the first three digits refer to the Data Dictionary tag, the last three digits refer to the Access Layer tag):

```
$ module load IMAS/3.22.0-4.0.2
```

To initialize the directory structure for a new local IMAS database (to be done only once), type:

```
$ imasdb [dbname]
```

where `dbname` can be anything and is the name of the database to make under ~/public/imasdb.

Unlike in the previous Acces Layer 3.x versions, there is no more default database environment. The database name, IMAS major version and user name must always be specified explicitly by calling `imas_open_env` or `imas_create_env`.

To view the Data Dictionary corresponding to the current environment, simply type:

```
$ dd_doc
```

This will launch the default web browser with an HTML description of the Data Model.

Try the example code in access layer/examples/dd-v3 to get started with compiling for imas. Be sure to use the Makefile-local.

5.2 Data Entries

A Data Entry is a collection of IDSs (potentially all IDSs) and their multiple occurrences (see 4.1). A Data Entry is uniquely defined by:

- IMAS major version (corresponds to the number of the Data Dictionary major revision)
- User name (or absolute path name)

- Database name
- Pulse number
- Run number

In the present implementation:

- IMAS major version can be any existing IMAS major version, at this moment : "3"
- User name is restricted to registered users on the system. If the User name starts with a "/" then it is interpreted as the absolute base path for the location of the IMAS data files (see below).
- Database name can be any name (e.g. iter or test), lower case
- Limitations in the MDS+ backend only:
    - Pulse number is limited to range 0-214748
    - Run number is limited to 5 digits : range 0-99999

5.3 Backend format for the stored data

With the new low level API (4.x), it's possible to store IMAS data in various backends and to choose which backend to use when creating or opening a new pulse file. Currently available backends are : MDSPLUS, HDF5, MEMORY, UDA and ASCII.

- `MDSPLUS_BACKEND`: Data Entry is stored in a MDSplus pulse files on disk
- `HDF5_BACKEND`: Data Entry is stored in HDF5 pulse files on disk
- `MEMORY_BACKEND`: Data Entry is stored in memory of the current process, no data is stored on disk, avoiding IO overheads
- `UDA_BACKEND`: Data Entry is accessed from UDA (see section 5.7 on UDA access)
- `ASCII_BACKEND`: Data Entry is stored in an ASCII file on disk (by default this file is written in the current directory with a name like "<dbname>_<shot>_<run>_<idsname>.ids"). This backend is so far tested only from the Fortran HLI. **Only PUT and GET are implemented** so far (no *_SLICE operation).

Note : the memory and mdsplus backends are now completely independent (this was not the case in the previous 3.x Access Layer versions), so the old enable_mem_cache, flush and discard functions are deprecated. The memory backend is not anymore a memory cache on top of an mdsplus file.

5.4 Organization of the local Data Files

5.4.1 MDS+ backend pulse files organization

The standard location on the user account is:

- For RUN numbers within 0-9999 : ~/public/imasdb/*DatabaseName*/*IMASMajorVersion*/0
- For RUN numbers within 10000-19999 : ~/public/imasdb/*DatabaseName*/*IMASMajorVersion*/1
- For RUN numbers within 20000-29999 : ~/public/imasdb/*DatabaseName*/*IMASMajorVersion*/2

- …
- For RUN numbers within 90000-99999 : ~/public/imasdb/*DatabaseName*/*IMASMajorVersion*/9

If the User name starts with a "/", then it is interpreted as the absolute base path for the location of the IMAS data files:

- For RUN numbers within 0-9999 : <Username>/imasdb/*DatabaseName*/*IMASMajorVersion*/0
- For RUN numbers within 10000-19999 : <Username>/imasdb/*DatabaseName*/*IMASMajorVersion*/1
- For RUN numbers within 20000-29999 : <Username>/imasdb/*DatabaseName*/*IMASMajorVersion*/2
- …
- For RUN numbers within 90000-99999 : <Username>/imasdb/*DatabaseName*/*IMASMajorVersion*/9

The present file names are (for the MDS+ backend):

- ids_*PulseRun*.tree
- ids_*PulseRun*.datafile
- ids_*PulseRun*.characteristics

Where *Pulse* is the pulse number and *Run* is the 4 rightmost digits of the run number of the Data Entry.

Example: PULSE 22, RUN 2 consists of 3 files:

- ids_220002.tree
- ids_220002.datafile
- ids_220002.characteristics

In principle, users do not need to access directly those files, since data operations should go through the Access Layer.

You can make your IMAS data available to other users by changing the access rights of the above directories and files.

5.4.2 HDF5 backend pulse files organization

For given shot and run numbers, a HDF5 master pulse file is created upon creation of a new IMAS data entry (see 5.5). At creation time, this file contains only the 2 attributes SHOT and RUN corresponding to the data entry. Each IDS occurrence is stored in a separate file, created by the IDS PUT operation (i.e. files exist only for the IDS that have been effectively PUT in the data entry). In addition, a reference to this file is appended in the master file. An example is given below for SHOT=9998 and RUN=9998 where the master pulse file references 4 separate IDS files: the edge_profiles IDS with occurrence 0 (pulse file edge_profiles.h5 file), two edge_transport IDSs with occurrence 0 (pulse file edge_transport.h5 file) and

occurrence 1 (pulse file edge_transport_1.h5 file) and the equilibrium IDS with occurrence 0 (pulse file equilibrium.h5 file).

All pulse files are located in the user's account under the folder:
~/public/imasdb/DatabaseName/IMASMajorVersion/SHOT/RUN

The HDF5 backend doesn't have a number of limitations that are specific to the MDS+ backend :

- no limitation in SHOT and RUN range (other than the length of UNIX file names)
- no limitation on the maximum number of IDS occurrences that can be used
- no limitation on the maximum size of static AoS

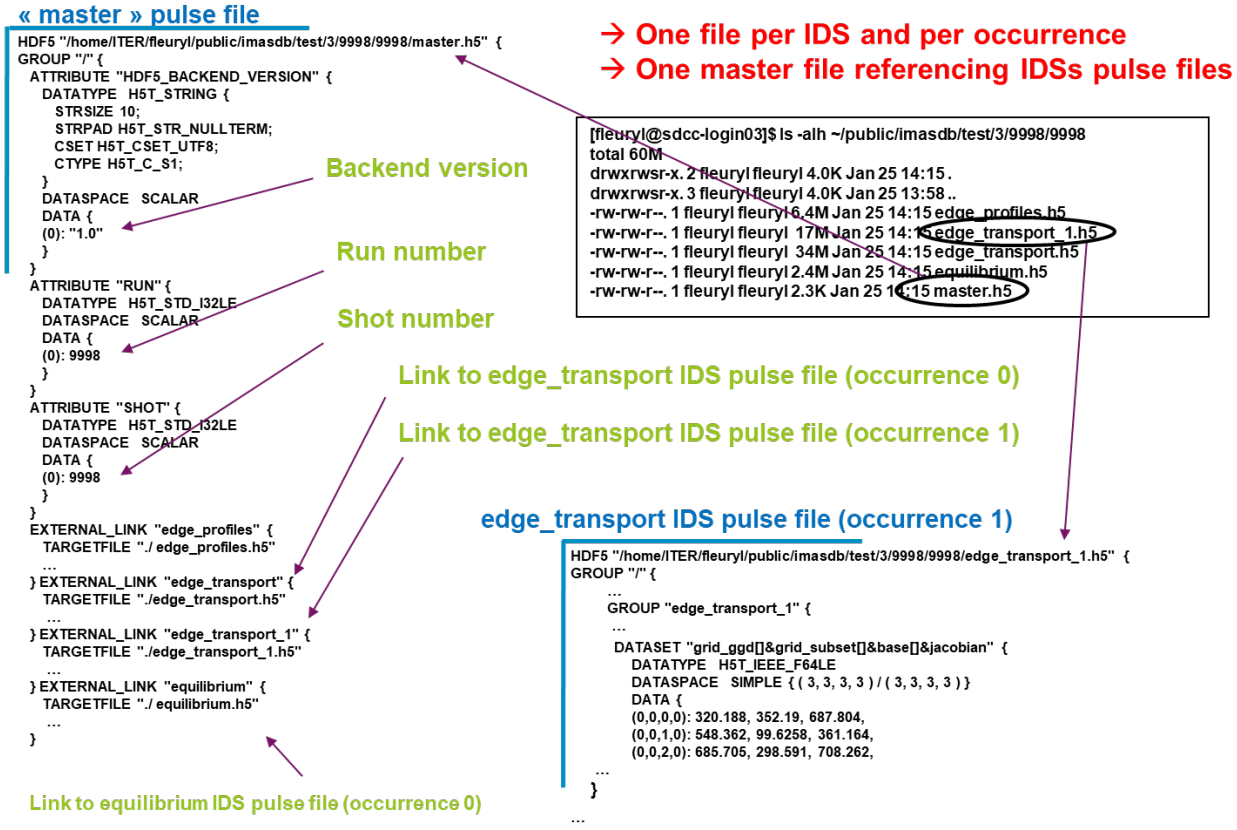

Figure 1: example of HDF5 backend files organization for shot=9998, run=9998

5.5 Creating a new Data Entry

A new Data Entry can be created by the following function (in this table and the following ones, input arguments as listed in green, output in blue):

- imas_create_env: for creating a new Data entry (uses by default the MDS+ backend). Be careful, this function erases the previous Data entry if it existed.

Language	***Syntax of the imas_create_env function***
Fortran	`call imas_create_env(treename,pulse,run,0,0,idx,UserName, DatabaseName,IMASMajorVersion,retstatus)` To select other backends, see 5.16

C++	Create IDS object `IdsNs::IDS ids(pulse,run,0,0);` then `int retstatus = createEnv(UserName, DatabaseName, IMASVersion);` on newly created object
	e.g.
	`IdsNs::IDS ids(12,1,0,0);`
	`ids.createEnv('usr', 'test', '3');`
	
	To set a different backend than MDS+, use setBackend(backend_id) before calling createEnv.
	
	For example, to use the memory backend:
	
	`ids.setBackend(MEMORY_BACKEND);`
	
	`ids.createEnv('usr', 'test', '3');`
	
	Other example, to use the HDF5 backend:
	
	`ids.setBackend(HDF5_BACKEND);`
	`ids.createEnv('usr', 'test', '3');`

Java	`imas.createEnv (pulse, run, UserName, DatabaseName, IMASMajorVersion, [BACKEND])`, for example :
	`idx = imas.createEnv(12, 1, "user", "test", "3") ;`
	`or, to use a different backend than MDS+:`
	`idx = imas.createEnv(12, 1, "user", "test", "3", LowLevel.MDSPLUS_BACKEND) ;`

Python	`import imas`
	`from imas import imasdef`
	
	`imas_entry = imas.DBEntry(Backend_id, DatabaseName, pulse, run)`
	`imas_entry.create()`
	
	Optional arguments of the DBEntry class:
	`user_name = UserName`
	`data_version = IMASMajorVersion`
	
	Optional arguments of the create function:
	`options = backend-specific options`
	
	`Backend_id` may be :

	ASCII : imasdef.ASCII_BACKEND MDS+ : imasdef.MDSPLUS_BACKEND HDF5 : imasdef.HDF5_BACKEND Memory : imasdef.MEMORY_BACKEND

Matlab	`idx = imas_create_env( treename, pulse, run, UserName, DatabaseName,IMASMajorVersion )`
	
	To use a different backend than MDS+:
	
	`idx = imas_create_env_backend(pulse, run, UserName, DatabaseName, IMASMajorVersion, backend_id)`
	
	For example, to use the HDF5 backend, set backend_id parameter to 13:
	
	`idx = imas_create_env_backend(pulse, run, UserName, DatabaseName, IMASMajorVersion, 13)`

`retstatus` is the returned error flag, 0 means that the operation was successful. Negative values indicate an exception has been raised, with the following meaning:

- -2 : a context exception has been raised
- -3 : a backend exception has been raised
- -4 : an abstract low level layer exception has been raised
- -1 : all other exceptions (unknown error)

`idx` is the returned identifier (integer), referencing the Data entry for subsequent data access commands (Get, Put, Close, …). In the Matlab interface only, in case of encountering an exception, `idx` is negative and contains the value of `retstatus` as described above.

After the `imas_create_env` command, the Data Entry remains open and accessible via its `idx` reference.

With this system, multiple Data Entries can remain opened at a given time.

5.6 Opening an existing Data Entry

An existing Data Entry can be opened by the following function:

- imas_open_env: for opening an existing Data entry (uses by default the MDS+ backend).

Language	***Syntax of the imas_open_env function***

Fortran	`call imas_open_env(treename,pulse,run,idx,UserName, DatabaseName,IMASMajorVersion,retstatus)` To select other backends, see 5.16

C++	Create IDS object `IdsNs::IDS imas_entry(pulse,run,0,0);` then `int retstatus = openEnv(UserName, DatabaseName, IMASMajorVersion)` on the newly created object
	e.g.
	`IdsNs::IDS imas_entry(12,1,0,0);`
	`imas_entry.openEnv('usr', 'test', '3');`
	
	To set a different backend than MDS+, add the following command before calling createEnv:
	
	`ids.setBackend(MEMORY_BACKEND);`
	
	`ids.openEnv('usr', 'test', '3');`

Java	`imas.openEnv(pulse, run, UserName, DatabaseName, IMASVersion, [BACKEND])`, for example :
	`idx = imas.openEnv(12, 1, "user", "test", "3") ;`
	or, to use a different backend than MDS+:
	`idx = imas.openEnv(12, 1, "user", "test", "3", LowLevel.MEMORY_BACKEND) ;`

Python	`import imas`
	`from imas import imasdef`
	
	`imas_entry = imas.DBEntry(Backend_id, DatabaseName, pulse, run)`
	`imas_entry.open()`
	
	Optional arguments of the DBEntry class:
	`user_name = UserName`
	`data_version = IMASMajorVersion`
	
	Optional arguments of the create function:
	`options = backend-specific options`
	
	`Backend_id` may be :
	ASCII : imasdef.ASCII_BACKEND
	MDS+ : imasdef.MDSPLUS_BACKEND
	HDF5 : imasdef.HDF5_BACKEND
	Memory : imasdef.MEMORY_BACKEND

Matlab	`idx = imas_open_env( treename, pulse, run, UserName, DatabaseName, IMASMajorversion )`
	
	`e.g.`
	
	`idx = imas_open_env('ids', 54178, 0, 'imas', 'test','3')`
	
	To use a different backend than MDS+:
	
	`idx = imas_open_env_backend(pulse, run, UserName, DatabaseName, IMASMajorVersion, backend_id)`
	
	For example, to use the HDF5 backend, set backend_id parameter to 13:
	
	`idx = imas_open_env_backend(pulse, run, UserName, DatabaseName, IMASMajorVersion, 13)`

`retstatus` is the returned error flag, 0 means that the operation was successful. Negative values indicate an exception has been raised, with the following meaning:

- -2 : a context exception has been raised
- -3 : a backend exception has been raised
- -4 : an abstract low level layer exception has been raised
- -1 : all other exceptions (unknown error)

`idx` is the returned identifier (integer), referencing the Data entry for subsequent data access commands (Get, Put, Close, …). In the Matlab interface only, in case of encountering an exception, `idx` is negative and contains the value of `retstatus` as described above.

With this system, multiple Data Entries can remain opened at a given time.

5.7 Opening a remote Data Entry using the UDA backend

This functionality allow to:

- Access remote IMAS data entries
- Access remote experimental data with on-the-fly mapping to IMAS format

A number of UDA plugins already exist for these, but their availability depends on how UDA has been installed on the local cluster. Therefore it's recommended that you contact the IMAS support team when you want to use this functionality.

Language	***Syntax of the imas_open_env function***
Fortran	`Not implemented yet`
C++	`Not implemented yet`
Java	`Not implemented yet`

Python	Instantiate a DBEntry variable with UDA as backend and the appropriate user_name for that experiment (usually 'public', but this is experiment-dependent):
	
	`import imas`
	`from imas import imasdef`
	
	`imas_entry =`
	`imas.DBEntry(imasdef.UDA_BACKEND,`
	`DatabaseName, pulse, run,`
	`user_name='public')`
	`imas_entry.open()`

Matlab	`idx =`
	`imas_open_public(MachineName,pulse,run)`
	
	`e.g.`
	
	`idx = imas_open_public('WEST',54178,0)`

5.8 Closing an opened Data Entry

An opened Data Entry can be closed by the following function:

Language	***Syntax of the imas_close function***
Fortran	`call imas_close(idx, retstatus)`

C++	Method `close()` of the imas_entry object.
	e.g.
	`IdsNs::IDS imas_entry(12,1,0,0);`
	`int retstatus = imas_entry.close();`

Java	`imas.close(idx) ;`
Python	def `close(self)` e.g. `retstatus = imas_entry.close()`

Matlab	`[ retstatus ] = imas_close(idx)`

`retstatus` is the returned error flag, 0 means that the operation was successful (see 5.5 for the meaning of the negative values, indicating an error has occurred)

`idx` is the Data Entry identifier returned by the Open or Create command.

5.9 Putting an IDS

Putting an IDS means writing the content of an IDS variable from a code/memory workspace into a Data Entry.

The IDS variable is PUT entirely, with all time slices it may contain. The PUT command deletes any previously existing data within the target IDS occurrence in the Data Entry.

The IDS variable can have none or many empty fields, they will simply be ignored by the PUT command if empty, leaving the fields also empty in the data entry. Some fields are nonetheless required to be filled before a PUT command, see 4.2.

Language	***Syntax of the ids_put function***
Fortran	`call ids_put(idx, IDSOccurrence, IDSVariable, retstatus)`

C++	Method of the ids object: def `put(self, occurrence=0)`
	Example:
	`IdsNs::IDS imas_entry(12,1,0,0);`
	`imas_entry._pf_active.ids_properties.comment_of = "Test data";`
	`imas_entry._pf_active.put();` for occurrence 0
	`imas_entry._pf_active.put(n);` for occurrence n

Java	Method of the ids: `put (idx, IDSOccurrence, IDSVariable)`, for example :
	`imas.equilibrium equilibrium = new imas.equilibrium();`
	`equilibrium.put(idx, "equilibrium", equilibrium) ;`

Python	Method of the imas_entry object:
	
	`imas_entry.put(ids, occurrence = 0)`
	where `ids = imas.<IDSname>` object (the IDS variable instantiating an IDS of type <IDSname>
	
	e.g.
	`ids = imas.pf_active()`
	`- Fill the IDS variable here -`
	`imas_entry.put(ids)`

Matlab	`ids_put(idx, IDSOccurrence, IDSVariable)`

`retstatus` is the returned error flag, 0 means that the operation was successful

`idx` is the Data Entry identifier returned by the Open or Create command.

`IDSOccurrence` is a string indicating the name and occurrence of the target IDS in the Data Entry (see 4.1)

`IDSVariable` is the IDS variable to be PUT

5.10 Putting an IDS progressively in a time loop

A frequent use case is the simulation of time evolution, in which one has to put the constant and static values only once and then progressively append the subsequent dynamic values calculated at each loop step. In such use case it is expected that the IDS variable contains one or more time slices to be appended to the already stored part of the IDS, in any non-empty dynamic node. It's thus possible to append several time slices to a node of the IDS in one PUT_SLICE call, however the user must ensure that the size of the time dimension of the node remains consistent with the size of its timebase. An initial call to the PUT function at the first iteration of the loop will first delete any previously existing data within the target IDS occurrence in the Data Entry and then put simultaneously the constant/static and the dynamic data of the IDS variable. Then the PUT_SLICE function should be called for every subsequent iteration of the loop, which will append the content of the non-empty dynamic nodes of the IDS variable to the time dimension of the nodes in the stored IDS.

Using PUT_SLICE without a preceding PUT is forbidden as this operation requires knowing which constant value was set for ids_properties/homogeneous_time node.

PUT_SLICE does not require anymore the IDS to be in homogeneous mode (previous limitation of Access Layer 3.x versions). Timebases with different values and sizes can be appended. By not allocating some dynamic nodes in the IDS variable for some iterations of the time loop (e.g. slowly-varying quantities), it is possible to skip appending to them a value, thus enabling to construct an IDS containing time bases with different sizes with a series of PUT_SLICE calls.

Time slices must be appended in strictly increasing time order, since the Access Layer is not reordering time arrays. Doing otherwise will result in non-monotonic time arrays, which will create confusion and make subsequent GET_SLICE commands to fail.

Although being put progressively time slice by time slice, the final IDS must be compliant with the data dictionary. A typical error when constructing IDS variables time slice by time slice is to change the size of the IDS fields during the time loop, which is not allowed but for the children of an array of structure which has time as its coordinate.

The PUT_SLICE command is appending data, so does not modify previously existing data within the target IDS occurrence in the Data Entry.

The IDS variable can have none or many empty fields, they will simply be ignored by the PUT_SLICE command if empty. Some fields are nonetheless required to be filled before a PUT_SLICE command, see 4.2.

Language	***Syntax of the ids_put_slice function***
Fortran	`call ids_put_slice(idx, IDSOccurrence, IDSVariable, retstatus)`
C++	Method of the ids object: def `putSlice(self, occurrence=0)` Example: `IdsNs::IDS imas_entry(12,1,0,0);` `imas_entry._actuator.putSlice();` for occurrence 0 `imas_entry._actuator.putSlice(n);` for occurrence n

Java	Method of the ids: `putSlice (idx, IDSOccurrence, IDSVariable)`, for example :
	`imas.equilibrium equilibrium = new imas.equilibrium();`
	`equilibrium. putSlice(idx, "equilibrium", equilibrium) ;`

Python	Method of the imas_entry object:
	`imas_entry.put_slice(ids, occurrence = 0)`
	where `ids = imas.<IDSname>` object (the IDS variable instantiating an IDS of type <IDSname>
	
	e.g.
	`ids = imas.pf_active()`
	`- Fill the IDS variable here -`
	`imas_entry.put_slice(ids)`

Matlab	`ids_put_slice(idx, IDOccurrence, IDSVariable)`

`retstatus` is the returned error flag, 0 means that the operation was successful

`idx` is the Data Entry identifier returned by the Open or Create command.

`IDSOccurrence` is a string indicating the name and occurrence of the target IDS in the Data Entry (see 4.1)

`IDSVariable` is the IDS variable to be PUT

5.11 Deleting an IDS in a Data Entry

Deleting an IDS in a Data Entry is done using the IDS_DELETE command.

Language	***Syntax of the ids_delete function***
Fortran	`call ids_delete(idx, IDSOccurrence, IDSVariable)`
C++	Method remove of the ids e.g. `ids._pf_active.remove()`

Java	Method of the ids: `delete (idx, IDSOccurrence, IDSVariable)`, for example :
	`imas.equilibrium equilibrium = new imas.equilibrium();`
	`equilibrium.delete(idx, "equilibrium", equilibrium) ;`

Python	Method of the imas_entry object:
	
	`imas_entry.delete_data(<IDSname>, occurrence = 0)`
	e.g.
	`imas_entry.delete_data("equilibrium", 1)`

Matlab	Not implemented yet

`idx` is the Data Entry identifier returned by the Open or Create command.

`IDSOccurrence` is a string indicating the name and occurrence of the target IDS in the Data Entry (see 4.1)

`IDSVariable` is an empty or filled IDS variable, used only to indicate to the Access Layer the type of IDS to be deleted. Its content does not matter.

5.12 Getting an IDS

Getting an IDS means reading the content of an IDS variable from a Data Entry into a code/memory workspace.

The GET IDS Data Entry command fetches an IDS in its entirety, with all time slices it may contain.

Empty fields within the IDS in the Data Entry are returned with the default values indicated in section 2.2.

Language	***Syntax of the ids_get function***
Fortran	`call ids_get(idx, IDSOccurrence, IDSVariable, retstatus)`
C++	Method of the ids def `get(self, occurrence=0)` Example: `IdsNs::IDS ids(12,1,0,0);` `ids.open();` //Open the database `ids._pf_active.get();` for occurrence 0 `ids._pf_active.get(n);` for occurrence n
Java	Method of the ids: `get(idx, IDSOccurrence)`, for example : `imas.equilibrium equilibrium = imas.equilibrium.get(idx, "equilibrium") ;`

Python	Method of the imas_entry object:
	`imas_entry.get(<IDSname>, occurrence = 0)`
	
	e.g.
	`ids = imas_entry.get("pf_active",1)`

Matlab	`IDSVariable = ids_get(expIdx, IDSOccurrence)`

`idx` is the Data Entry identifier returned by the Open or Create command.

`IDSOccurrence` is a string indicating the name and occurrence of the target IDS in the Data Entry (see 4.1)

`IDSVariable` is the IDS variable to be filled by the GET

5.13 Getting a time slice from an IDS

A frequent Use Case, instead of considering a full evolution of the input, is to consider only a time snapshot of the physics quantities. The GET_SLICE command allows to obtain a single time slice extracted from an IDS in a Data Entry, at a requested time slice, using a given time interpolation method since it cannot be guaranteed that a time slice corresponding exactly to the requested time exists in the original IDS.

The GET_SLICE command returns an IDS Variable with the constant/static data and the dynamic data interpolated on the requested time slice (i.e. the size of the time dimension of the dynamic data is one).

Empty fields within the IDS in the Data Entry are returned with the default values indicated in section 2.2.

Language	***Syntax of the ids_get_slice function***	***Available constants for interpolation_method (can be used instead of the numerical value)***

Fortran	`call ids_get_slice(idx, IDSOccurrence, IDSVariable, time_requested, interpolation_method, retstatus)`	`CLOSEST_INTERP=1`
		`PREVIOUS_INTERP=2`
		`LINEAR_INTERP=3`

C++	Method of the ids def `getSlice(self, occurrence=0, time_requested, interpolation_method)` Example: `IdsNs::IDS ids(12,1,0,0);` `ids._pf_active.getSlice(time_requested, interpolation_method);` for occurrence 0 `ids._pf_active.getSlice(n, time_requested, interpolation_method);` for occurrence n	CLOSEST_SAMPLE PREVIOUS_SAMPLE INTERPOLATION
Java	Method of the ids: `getSlice(idx, IDSOccurrence, double time_requested, int interpolation_method)`, for example : `imas.equilibrium equilibrium = imas.equilibrium.getSlice(idx, "equilibrium", 0.1, 2) ;`	CLOSEST_SAMPLE PREVIOUS_SAMPLE INTERPOLATION,

Python	Method of the imas_entry object:
	`imas_entry.get_slice(<IDSname>, time_requested, interpolation_method, occurrence = 0)`
	
	e.g.
	`ids = imas_entry.get_slice("pf_active",1.3,imasdef.LINEAR_INTERP)`
		imasdef.py:
		UNDEFINED_INTERP
		CLOSEST_ INTERP
		PREVIOUS_INTERP
		LINEAR_INTERP
		
		e.g.
		`interpolation_method= imasdef.LINEAR_INTERP`

Matlab	`IDSVariable = ids_get_slice(idx, IDSOccurrence, time_requested, interpolation_method)`	

`retstatus` is the returned error flag, 0 means that the operation was successful

`idx` is the Data Entry identifier returned by the Open or Create command.

`IDSOccurrence` is a string indicating the name and occurrence of the target IDS in the Data Entry (see 4.1)

`IDSVariable` is the IDS variable to be filled by the GET

`time_requested` is the time (scalar) that is requested

The following interpolation methods are implemented (others may be implemented in the future if needed):

- Interpolation_method = 1: returns the closest time slice in the original IDS
- Interpolation_method = 2: returns the previous time slice in the original IDS
- Interpolation_method = 3: linear interpolation between the previous and the next time slice

Note that the linear interpolation will be successful only if, between the two time slices of an interpolated dynamic array of structure, the same leaves are populated and they have the same size. Otherwise the GET_SLICE call will be canceled and an error reported.

5.14 Getting the units of a node

Available in Python only so far, the following function returns a string containing the units of a node.

Language	***Syntax of the function***

Python	`from imas.dd_units import DataDictionaryUnits`
	`dd = DataDictionaryUnits()`
	`dd.get_units(<IDSname>,node_path)`
	`example :`
	`dd.get_units('waves','time')` will return “s”

`<IDSname>` – IDS name
`node_path` – path, relative to the IDS root node, to the requested node .

- The path to the requested node is separated by slashes (“/path/to/node”).
- Array of structure indices must be ignored when typing the path (since the units will be the same for all indices)

The function can also be called from command line:

```
./idsunits.py --ids waves --field time
```

The units of the whole Data Dictionary are processed during the first call to this function and the results cached in the user local disk space. Subsequent calls will thus be much faster.

5.15 Getting a subset of an IDS

Getting a subset of an IDS enables reading only a node (and its descendants if the node is a structure), making the GET operation much faster.

To retrieve only requested node one should call the method partialGet. As it depends heavily on object introspection (i.e. the ability to determine the type and fields of an object at runtime) it has been implemented only in Python so far.

Language	***Syntax of the function***

Python	Method of the imas_entry object:
	`imas_entry.partial_get(<IDSname>, dataPath, occurrence = 0)`
	
	e.g.
	`ids = imas_entry.partial_get("magnetics","flux_loop(:)/voltage/data")`

`dataPath` - specifies the path, relative to the IDS root node, to the requested node .
Path syntax:

- The path to requested node(s) is separated by slashes ("/path/to/node(s)").
- Nodes representing arrays must contain indexes ("/path/to/array(idx)/field") or "Fortran style" indices ("path/to/array(x:y)/field")
- Limitation: In case of nested AoS, it is not allowed to specify set of indices for AoS ancestors. Only given values of AoS ancestors indices are handled: (e.g. "field/with/ancestorAoS(x:y)/field/AoS(n :m)" is not managed)

Data query examples:

- "`flux_loop(1)/flux/data(1:5)`"
- "`bpol_probe(2:3)/field/data`"
- "`loop(:)/current`"
- "`time(4:-1)`"
- "`profiles_1d(2)/grid/rho_tor_norm(2:4)`"
- "`ids_properties`"

`occurrence` is a number indicating the occurrence of the target IDS in the Data Entry

Returned data:

- During the partial GET, only the requested node is returned (and the structure below if the node is not a leaf), not the whole IDS structure above
- Reshaping of returned data:
    - If the user specifies a set of indices for an Array of Structure parent of the requested node, then the partialGet returns a reshaped leaf with an additional dimension (corresponding

to the one of its parent). This is doable only if the requested node has the same size for all requested AoS parent indices, otherwise the method will return an error
    - Reshaping is done by keeping time as the last dimension, to keep consistent with the IMAS DD rule (even though the reshaped leaf is not anymore a legal DD entity)
- Time issues:
    - If the IDS has been filled in homogeneous time mode, the values of the global time ids/time are copied to each time base of the returned structure

5.16 Selecting the backend by direct low level calls

Some HLIs don't allow yet to select a different backend than MDS+. Below are the direct low level calls to be carried out for selecting backend and opening a pulse file using direct lowlevel calls (use only if you don't have the option to do it from the HLI).

Note that for all functions returning a status, this status is an integer and `status=0` means success.

5.16.1 Selecting the Data Entry

To select the Data Entry, backend included, one should call the ual_begin_pulse_action function

Language	***Syntax of the ual_begin_pulse_action function***
Fortran	`call ual_begin_pulse_action( backend_id, pulse, run, UserName, DatabaseName, IMASMajorVersion, idx)`
C++	`idx = ual_begin_pulse_action( backend_id, pulse, run, UserName, DatabaseName, IMASMajorVersion)`
Java	`pulseCtx = LowLevel.ual_begin_pulse_action(backend_id, pulse, run, UserName, DatabaseName, IMASMajorVersion`
Python	`_ual_lowlevel.ual_begin_pulse_action(backend_id, pulse, run, UserName, DatabaseName, IMASMajorVersion)`
Matlab	

`idx` is the Data Entry identifier that needs to be passed to further functions like ual_open_pulse and ual_close_pulse

`backend_id` is an integer that identifies the backend. Constants are defined for each available backend and should be used as `backend_id` argument rather than their numerical value:

- `MDSPLUS_BACKEND`: Data Entry is stored in a MDSplus pulse files on disk

- MEMORY_BACKEND: Data Entry is stored in memory of the current process, no data is stored on disk, avoiding IO overheads
- UDA_BACKEND: Data Entry is accessed from UDA (see section 5.7 on UDA access)
- ASCII_BACKEND: Data Entry is stored in an ASCII file on disk (by default this file is written in the current directory with a name like "<dbname>_<shot>_<run>_<idsname>.ids"). This backend is so far tested only from the Fortran HLI. **Only PUT and GET are implemented** so far (no *_SLICE operation).

Note : the memory and mdsplus backends are now completely independent (this was not the case in the previous 3.x Access Layer versions), so the old enable_mem_cache, flush and discard functions are deprecated. The memory backend is not anymore a memory cache on top of an mdsplus file.

5.16.2 Opening the access to the selected Data Entry

Once the Data Entry has been selected (see section 5.16.1), its access needs to be granted by using the function ual_open_pulse, given a chosen mode of access (see below).

Language	***Syntax of the ual_open_pulse function***
Fortran	`call ual_open_pulse(idx, mode, options, status)`
C++	`status = ual_open_pulse(idx, mode, options)`
Java	`LowLevel.ual_open_pulse(idx, mode, options);`
Python	`ual_lowlevel.ual_open_pulse(idx, mode, options)`
Matlab	

idx is the Data Entry identifier that needs to be passed to further functions like ids_get, ids_put, ual_close_pulse, etc…

options is a string that allows to specify any backend specific additional options, pass empty string if no options are needed/available

- Options for ASCII_BACKEND:
  - –prefix <string>: adds a prefix to the target filename
  - –suffix <string>: adds a suffix to the target filename
  - –fullpath <string>: fully specify path for the target filename (allows for instance to specify /dev/stdin or /dev/stdout)

mode is an integer that identifies the different modes of access for a Data Entry (note that read and/or write access are granted depending on the access right restrictions of the targeted media, for instance the file permissions). Constants are defined for available modes:

- OPEN_PULSE: opens the access to the data only if the Data Entry exists, returns error otherwise

- FORCE_OPEN_PULSE: opens access to the data, creates the Data Entry if it does not exists yet
- CREATE_PULSE: creates a new empty Data Entry (returns error if Data Entry already exists) and opens it at the same time
- FORCE_CREATE_PULSE: creates an empty Data Entry (overwrites if Data Entry already exists) and opens it at the same time

5.16.3 Closing the access to the selected Data Entry

To stop using a Data Entry, its access should be closed using the ual_close_pulse function (see below)

Language	***Syntax of the ual_open_pulse function***
Fortran	`call ual_close_pulse(idx, mode, options, status)`
C++	`status = ual_close_pulse(idx, mode, options)`
Java	`LowLevel.ual_close_pulse(idx, mode, options);`
Python	`ual_lowlevel.ual_close_pulse(idx, mode, options)`
Matlab	

`idx` is the Data Entry identifier; after a call to `ual_close_pulse`, only two functions are allowed on this `idx`: ual_open_pulse, to interact further with the Data Entry, and `ual_end_action(idx, status)` to clean the lowlevel object associated with this identifier (after which the `idx` should not be used anymore as input argument of the access-layer API)

`options` is a string which allows to specify any backend specific additional options, pass empty string if no options are needed/available (so far the existing backends don't use these options))

`mode` is an integer which identifies the different ways to close the access of a Data Entry. Constants are defined for available modes:

- CLOSE_PULSE: close the access to the Data Entry
- ERASE_PULSE: close the access to the Data Entry and delete the underlying file, if the associated backend is file-based, and given sufficient permissions on the target file(s).

5.17 HDC Access Layer

The HDC access layer provides an alternative way of accessing data from the IDS data files through the Hierarchical Data Container (HDC) library. The documentation for this library can be found at http://compass-tokamak.pages.tok.ipp.cas.cz/HDC/ with examples given below for how to use this library to access and manipulate HDC IMAS data trees. When using the HDC Access Layer, the IDS variable in memory is no longer one of the native data types defined in section 2.2, but an HDC object that has to be manipulated via the HDC library.

5.17.1 API overview

Importing the library
Before you can start using the library you may need to import the module/headers.

Language	***Import declarations***
Fortran	`use al_hdc`
C++	`#include <al_hdc.h>`
Java	`import org.iter.imas.al_hdc;`
Python	`import al_hdc`
Matlab	`N/A`

Open

The open method is used to access an existing IDS file. It will return an error if no file exists with the given URI.

The URI syntax is still in development and is subject to change but the form currently used by the HDC access layer is:

imas:///[database]?query_args

- database: A placeholder for being able to choose different databases. Currently has to be set to “iter”.
- query_args: The query args used to select the data entry to open/create, i.e. machine=iter&shot=123&run=0. The possible query args are:
    - machine: The machine name for the data entry.
    - shot: The shot number of the data entry.
    - run: The run number of the data entry.
    - user: The user to access the data for (defaults to the current user).
    - imas_version: The IMAS major version to use (defaults to “3”).

Language	***Syntax of the open method***

Fortran	`integer :: idx`
	`integer :: status`
	`idx = open_entry(“imas:///iter?machine=test&shot=1&run=0”, status)`

C++	`imas::DataEntry entry{“imas:///iter?machine=test&shot=1&run=0”};` `entry.open();`

Java	`DataEntry entry = new DataEntry("imas:///iter?machine=test&shot=1&run=0"); entry.open()`
Python	`entry = al_hdc.DataEntry("imas:///iter?machine=test&shot=1&run=0") entry.open()`
Matlab	`entry = IMAS_HDC("imas:///iter?machine=test&shot=1&run=0") entry.open()`

Create

The create method is used to create a new IDS file. It will return an error if an existing file with the given URI already exists.

Language	***Syntax of the create method***
Fortran	`integer :: idx integer :: status idx = create_entry("imas:///iter?machine=test&shot=1&run=0", status)`
C++	`imas::DataEntry entry{"imas:///iter?machine=test&shot=1&run=0"}; entry.create();`
Java	`DataEntry entry = new DataEntry("imas:///iter?machine=test&shot=1&run=0"); entry.create()`
Python	`entry = al_hdc.DataEntry("imas:///iter?machine=test&shot=1&run=0") entry.create()`
Matlab	`entry = IMAS_HDC("imas:///iter?machine=test&shot=1&run=0") entry.create()`

Close

The close method closes an open IDS file.

Language	***Syntax of the close method***
Fortran	`integer :: idx integer :: status call close_entry(idx, status)`

C++	`entry.close()`
Java	`entry.close()`
Python	`entry.close()`
Matlab	`entry.close()`

Get

The get method is used to read all or part of an IDS structure from an open file.

Language	***Syntax of the get method***

Fortran	`type(hdc_t) :: ids`
	`integer :: idx`
	`integer :: status`
	`status = get_ids(idx, "path", ids, occurrence)`
	
	`Note: occurrence is optional and defaults to 0`

C++	`imas::IDS ids = entry.get("path", occurrence=0)`

Java	`HDC ids = entry.get("path", occurrence)`
	
	`HDC ids = entry.get("path")`
	
	`Note: if the function is called without occurrence then the default value of 0 is used.`

Python	`ids = entry.get("path", occurrence)`
	
	`Note: if the function is called without occurrence then the default value of 0 is used.`

Matlab	`ids = entry.get("path", occurrence)`

Get Slice

The get_slice method is used to read a given time slice through an entire or part of an IDS structure.

Language	***Syntax of the get_slice method***

Fortran	`type(hdc_t) :: ids`
	`integer :: idx`
	`integer :: status`

	`status = get_ids_slice(idx, "path", time, interp_mode, ids, occurrence)`
	
	`Note: interp_mode is optional and defaults to 0, occurrence is optional and defaults to 0`

C++	`imas::IDS ids = entry.get_slice("path", slice_time, interp_mode=ualconst::closest_interp, occurrence=0)`

Java	`HDC ids = entry.get_slice("path", slice_time, interp_mode, occurrence)`
	
	`HDC ids = entry.get_slice("path", slice_time, interp_mode)`
	
	`Note: if the function is called without occurrence then the default value of 0 is used.`

Python	`ids = entry.get_slice("path", slice_time, interp_mode, occurrence)`
	
	`Note: if the function is called without occurrence then the default value of 0 is used.`

Matlab	`ids = entry.get("path", occurrence, slice_time, interp_mode, occurrence)`

Put

Put an IDS structure. This can be done either at the root node if the root of the given structure is an IDS (i.e. magnetics, equilibrium, etc.) or at a specified IDS path if the given structure is a sub-tree of the IDS structure.

By default the `put` function clears the existing data in the IDS data file, from the IDS root when putting the full IDS or from the node (and below if this is a structure) corresponding to the path indicated in case of a partial_put. If you wish to put without clearing the existing data then the clear_ids flag can be used – passing false for this flag will disable the clearing of data on the put (full or partial).

Warning : partial put operations may create inconsistencies in the stored IDS (either physical inconsistencies or structural inconsistencies in the Data Dictionary sense, e.g. shape conflicts) and should be used very carefully. For example, if you PUT a node, make sure to PUT also its coordinates and to use consistent sizes.

Language	***Syntax of the put method***

Fortran	type(hdc_t) :: ids integer :: idx integer :: occurrence logical :: clear_ids integer :: status status = put_ids(idx, "path", ids, occurrence, clear_ids) Note: occurrence is optional and defaults to 0, clear_ids is optional and defaults to .true.
C++	imas::IDS ids; entry.put("path", ids, occurrence=0, clear_ids=true);

Java	HDC ids = new HDC();
	
	entry.put("path", ids);
	
	entry.put("path", ids, occurrence);
	
	entry.put("path", ids, occurrence, clear_ids);
	
	Note: if the function is called without occurrence then the default value of 0 is used, if the function is called without clear_ids then the default value of true is used.

Python	ids = HDC() entry.put("path", ids, occurrence, clear_ids); Note: if the function is called without occurrence then the default value of 0 is used, if the function is called without clear_ids then the default value of True is used.
Matlab	entry.put("path", ids, occurrence, clear_ids)

Put Slice

You can use the put_slice method to put an IDS data tree containing data for a specified slice time.

Language	***Syntax of the put_slice method***
Fortran	type(hdc_t) :: ids integer :: idx

	integer :: status status = put_ids_slice(idx, "path", ids, occurrence) Note: occurrence is optional and defaults to 0
C++	imas::IDS ids; entry.put_slice("path", ids, occurrence=0);

Java	HDC ids = new HDC();
	
	entry.put_slice("path", ids, occurrence);
	
	entry.put_slice("path", ids);
	
	Note: if the function is called without occurrence then the default value of 0 is used

Python	ids = HDC() entry.put_slice("path", ids, occurrence) Note: if the function is called without occurrence then the default value of 0 is used.
Matlab	entry.put_slice("path", ids, occurrence)

Put error handling

When an HDC data structure is passed to `put` or `put_slice` the structure of the tree is validated against the IDS data structure. If an error occurs during this validation (e.g. the HDC variable contains fields that are not part of the IDS data structure, or is missing mandatory fields such as homogeneous_time), then it will be returned to the calling program via different mechanisms depending on the language. The table below shows the different ways of catching these errors.

You can also call the `validate` function in place of the `put` function in order to run the validation checks but without putting any data to the IDS data structure. This can be useful when doing development in an interactive language such as Python or Matlab. The validate function has the same arguments as `put_slice`, i.e. `validate(path, ids, occurrence)`. Occurrence here is mandatory in the case of a partial PUT (with some data on the disk already) to check the existence of parameters such as `homogeneous_time` that must exist either in the provided hdc tree or in the existing IDS data structure.

Language	***Put error handling***
Fortran	integer :: status status = put_ids(…) if status .ne. 0

	then print *, "failed to put with code", status endif

C++	try {
	  entry.put(…);
	} catch (imas::ValidationError& ex) {
	  std::cerr <<
	}

Java	try {
	  entry.put(…);
	} catch (UALException ex) {
	  System.out.println(ex.getMessage())
	}

Python	try:
	  entry.put(…)
	except UALError as err:
	  print(err)

Matlab	entry.put(…) % error will be printed to console

5.17.2 Accessing IDS data using HDC access layer

The HDC access layer can be used to read IDS data that has been written by any of the existing access layers, or through the HDC access layer. To do so you need to follow the steps (with more details for each step given below):

1. Open the file
2. Use the 'get' or 'get_slice' function to load the desired data from the file.

Opening the file

The open method takes a URI which identifies the IMAS data file. This URI is used to open the file or will return an error if the file does not exist.

Getting the IDS data

The get method on an open IMAS data file is used to read all or part of an IDS data tree. The string which specifies the path uses the "/" character to mark the nodes of the data tree with "[]" to select elements of the tree (the index is 0 based), i.e.

- "magnetics" selects the entire magnetics IDS
- "magnetics/flux_loop" selects all the magnetics flux loops
- "magnetics/flux_loop[0]" selects the first flux loop
- "magnetics/flux_loop[0:3]" selects the first to fourth flux loop

The path syntax is the same as syntax used by the HDC library for accessing elements in an HDC data tree (see https://compass-tokamak.pages.tok.ipp.cas.cz/HDC/#path-syntax).

Regardless of the path used the data is always returned in the same way – an HDC data tree. This root of the tree will be the IDS path that was specified by the path and all the data contained below this node in the IDS structure will be retuned in the data tree. You can see how to query and modify the returned data tree in the section below.

If you are after an individual time slice through the IDS data rather than all available data you can use the 'get_slice' method instead, providing the slice time required and the interpolation method to use.

Examples of getting data in the different language bindings are given below.

```
#include <al_hdc.h>

int main()
{
    imas::DataEntry entry
        {"imas:///iter?machine=test&shot=1000&run=0"};
    entry.open();

    // Reads the entire equilibrium IDS
    imas::IDS equilibrium = entry.get("equilibrium");

    // Reads the first flux loop IDS structure
    imas::IDS fl1 = entry.get("magnetics/flux_loop[0]");

    // Reads a single value from the IDS
    imas::IDS fl3_r = entry.get("magnetics/flux_loop[2]/position/r");

    // Reads flux_loop data for specified time slice
    imas::IDS fl1_t10 = entry.get_slice("magnetics/flux_loop[0]",
        10.0, ualconst::linear_interp);

    entry.close();
    return 0;
}
```

C++ example

```
import al_hdc

entry = al_hdc.DataEntry("imas:///iter?machine=test&shot=1000&run=0")
entry.open()

# Reads the entire equilibrium IDS
equilibrium = entry.get("equilibrium")
```

```
# Reads the first flux loop IDS structure
fl1 = entry.get("magnetics/flux_loop[0]")

# Reads a single value from the IDS
fl3_r = entry.get("magnetics/flux_loop[2]/position/r")

# Reads flux_loop data for specified time slice
fl1_t10 = entry.get_slice("magnetics/flux_loop[0]", 10.0,
        al_hdc.InterpMode.LINEAR)

entry.close()
```

Python example

```
program main
  use hdc_fortran
  use al_hdc
  use al_hdc_defs
  implicit none

  integer :: idx
  integer :: status
  type(hdc_t) :: equilibrium, fl1, fl3_r, f11_t10

  idx = open_entry("imas:///iter?machine=test&shot=1000&run=0",
status)

  ! Reads the entire equilibrium IDS
  status = get_ids(idx, "equilibrium", equilibrium)

  ! Reads the first flux loop IDS structure
  status = get_ids(idx, "magnetics/flux_loop[0]", f11)

  ! Reads a single value from the IDS
  status = get_ids(idx, "magnetics/flux_loop[2]/position/r", fl3_r)

  ! Reads flux_loop data for specified time slice
  status = get_ids_slice(idx, "magnetics/flux_loop[0]", 10.0,
LINEAR_INTERP, f11_t10)

  call close_entry(idx, status)
end program main
```

Fortran example

```
entry = IMAS_HDC("imas:///iter?machine=test&shot=1000&run=0");
entry.open();

% Reads the entire equilibrium IDS
```

```
equilibrium = entry.get("equilibrium");

% Reads the first flux loop IDS structure
fl1 = entry.get("magnetics/flux_loop[0]");

% Reads a single value from the IDS
fl3_r = entry.get("magnetics/flux_loop[2]/position/r");

% Reads flux_loop data for specified time slice
fl3_r = entry.get_slice("magnetics/flux_loop[0]", 10.0,
        IMAS_HDC.LINEAR_INTERP);

entry.close();
```

Matlab example

```
import org.iter.imas.al_hdc.DataEntry;
import org.iter.imas.al_hdc.Constants;
import dev.libhdc.HDC;

class Example
{
    public static void main(String args[]) {
        DataEntry entry =
          new DataEntry("imas:///iter?machine=test&shot=1000&run=0");
        entry.open();

        // Reads the entire equilibrium IDS
        HDC equilibrium = entry.get("equilibrium");

        // Reads the first flux loop IDS structure
        HDC fl1 = entry.get("magnetics/flux_loop[0]");

        // Reads a single value from the IDS
        HDC fl3_r = entry.get("magnetics/flux_loop[2]/position/r");

        // Reads flux_loop data for specified time slice
        HDC fl1_t10 = entry.get_slice("magnetics/flux_loop[0]",
            10.0, Constants.LINEAR_INTERP);

        entry.close();
    }
}
```

Java example

5.17.3 Reading data from the HDC tree

Accessing the data from the IDS data tree populated using the get or get_slice methods requires the use of the HDC library. Documentation for this library can be found at http://compass-tokamak.pages.tok.ipp.cas.cz/HDC/index.html with some examples given below.

```
#include <al_hdc.h>

int main()
{
    imas::DataEntry entry
        {"imas:///iter?machine=test&shot=1000&run=0"};
    entry.open();

    imas::IDS magnetics = entry.get("magnetics");

    imas::IDS flux_loops = magnetics["flux_loop"];
    std::string json = flux_loops.serialize("json");

    imas::IDS flux_loop_3 = magnetics["flux_loop[2]"];
    imas::IDS fl3_data = flux_loop_3["flux/data"];

    assert(fl3_data.get_type() == HDC_DOUBLE);
    std::vector<size_t> shape = fl3_data.get_shape();
    assert(shape.size() == 1);
    double* data = fl3_data.as<double>();

    std::cout << shape[0] << std::endl;
    assert(shape[0] >= 10);
    for (int i = 0; i < 10; ++i) {
        std::cout << data[i] << std::endl;
    }

    return 0;
}
```

C++ example

```
import al_hdc

entry = al_hdc.DataEntry("imas:///iter?machine=test&shot=1000&run=0")
entry.open()

magnetics = entry.get("magnetics")

flux_loops = magnetics["flux_loop"]
json = flux_loops.serialize("json")

flux_loop_3 = magnetics["flux_loop[2]"]
fl3_data = flux_loop_3["flux/data"]
```

```
data = fl3_data.to_python() # returns numpy array
print(data.shape)
print(data)
```

Python example

```
program main
  use hdc_fortran
  use al_hdc
  use iso_c_binding
  implicit none

  integer :: idx
  integer :: status
  type(hdc_t) :: magnetics, flux_loop_3, fl3_data
  real(kind=dp), pointer :: data(:)
  integer(kind=c_long), allocatable :: shape(:)
  integer(kind=c_size_t) :: rank

  idx = open_entry("imas:///iter?machine=test&shot=1000&run=0",
status)

  status = get_ids(idx, "magnetics", magnetics)

  flux_loop_3 = hdc_get_child(magnetics, "flux_loop[2]")
  fl3_data = hdc_get_child(flux_loop_3, "flux/data")

  rank = hdc_get_rank(fl3_data)
  shape = hdc_get_shape(fl3_data)

! hdc_get will 'stop' if the data doesn’t match the dimension
! of the result array
  call hdc_get(fl3_data, data)

  print *, 'rank: ', rank
  print *, 'shape: ', shape(1)
  print *, 'data: ', data(1:10)
end program main
```

Fortran example

```
entry = IMAS_HDC("imas:///iter?machine=test&shot=1000&run=0");
entry.open();

magnetics = entry.get("magnetics");

flux_loop_3 = magnetics.get_child("flux_loop[2]");
fl3_data = flux_loop_3.get_child("flux/data");
```

```
data = fl3_data.get_data();
disp(data);
```

Matlab example

```
import org.iter.imas.al_hdc.DataEntry;
import dev.libhdc.HDC;
import org.nd4j.linalg.api.ndarray.INDArray;

class Example
{
    public static void main(String args[]) {
        DataEntry entry =
          new DataEntry("imas:///iter?machine=test&shot=1000&run=0");
        entry.open();

        HDC magnetics = entry.get("magnetics");

        HDC flux_loops = magnetics["flux_loop"];
        String json = flux_loops.serialize("json");

        HDC flux_loop_3 = magnetics["flux_loop[2]"];
        HDC fl3_data = flux_loop_3["flux/data"];

        assert fl3_data.get_type() == HDC_DOUBLE;
        ArrayList<Integer> shape = fl3_data.get_shape();
        assert shape.size() == 1;
        INDArray data = fl3_data.data();

        System.out.print(shape.toString());
        assert shape[0] >= 10;
        for (int i = 0; i < 10; i++) {
            System.out.print(data[0]);
        }
    }
}
```

Java example

5.17.4 Modifying an existing HDC data tree

After you have retrieved the HDC tree containing the IDS data you can modify the values in the tree using the HDC API. This allows for an HDC tree to be read from one IDS data entry, modified and written to a new IDS data entry.

```
#include <al_hdc.h>
```

```
int main()
{
    imas::DataEntry entry
        {"imas:///iter?machine=test&shot=1000&run=0"};
    entry.open();

    imas::IDS magnetics = entry.get("magnetics");

    imas::IDS fl3_data = magnetics["flux_loop[2]/flux/data"];
    std::vector<size_t> shape = fl3_data.get_shape();
    double* data = fl3_data.as<double>();

    for (int i = 0; i < shape[0]; ++i) {
        data[i] *= 2.0;
    }

    fl3_data.set_data(shape, data);

    return 0;
}
```

C++ example

```
import al_hdc

entry = al_hdc.DataEntry("imas:///iter?machine=test&shot=1000&run=0")
entry.open()

magnetics = entry.get("magnetics")

fl3_data = magnetics["flux_loop[2]/flux/data"]

data = fl3_data.to_python() # returns numpy array
data *= 2.0

fl3_data.set_data(data)
```

Python example

```
program main
  use hdc_fortran
  use al_hdc
  use iso_c_binding
  implicit none

  integer :: idx
  integer :: status
  type(hdc_t) :: magnetics, fl3_data
  real(kind=dp), pointer :: data(:)
```

```
  integer(kind=c_long), allocatable :: shape(:)
  integer :: i

  idx = open_entry("imas:///iter?machine=test&shot=1000&run=0",
status)

  status = get_ids(idx, "magnetics", magnetics)

  fl3_data = hdc_get_child(magnetics, "flux_loop[2]/flux/data")
  shape = hdc_get_shape(fl3_data)

! hdc_get will 'stop' if the data doesn't match the dimension
! of the result array
  call hdc_get(fl3_data, data)

  do i = 1, shape(0)
    data(i) = data(i) * 2.0
  end do

  call hdc_set(fl3_data, data)
end program main
```

Fortran example

```
entry = IMAS_HDC("imas:///iter?machine=test&shot=1000&run=0");
entry.open();

magnetics = entry.get("magnetics");

fl3_data = magnetics.get_child("flux_loop[2]/flux/data");

data = fl3_data.get_data();
data = data * 2.0;

fl3_data.set_data(data);
```

Matlab example

```
import org.iter.imas.al_hdc.DataEntry;
import dev.libhdc.HDC;
import org.nd4j.linalg.api.ndarray.INDArray;

class Example
{
    public static void main(String args[]) {
        DataEntry entry =
          new DataEntry("imas:///iter?machine=test&shot=1000&run=0");
        entry.open();
```

```
        HDC magnetics = entry.get("magnetics");

        HDC fl3_data = magnetics["flux_loop[2]/flux/data"];
        ArrayList<Integer> shape = fl3_data.get_shape();
        INDArray data = fl3_data.data();

        for (int i = 0; i < shape[0]; ++i) {
            data[i] *= 2.0;
        }

        fl3_data.set_data(shape, data);
    }
}
```

Java example

5.17.5 Creating a new HDC IDS data tree

Creating a new IDS to put requires the use of the HDC API. You first need to create a new empty HDC tree and then populate the nodes of the tree that correspond with the IDS data dictionary. The "ids_properties/homogeneous_time" element is mandatory and must be populated on each IDS.

An HDC data tree may contain multiple IDS structures (i.e. you can populate the "magnetics" and "equilibrium" IDSs in a single HDC tree) but in this case each IDS much be at the top of the tree.

```
#include <al_hdc.h>
#include <algorithm>

int main()
{
    imas::IDS tree;
    imas::IDS mag = tree["magnetics"];

    mag["ids_properties/homogeneous_time"] = 0;

    std::vector<double> time(100);
    std::vector<double> data(100);
    std::generate(time.begin(), time.end(),
                  [n=0.0]() mutable { n += 0.1; return n; });
    std::fill(data.begin(), data.end(), 100.0);

    for (int i = 0; i < 10; ++i) {
        std::string path = "flux_loop[" + std::to_string(i) + "]";
        mag[path + "flux/position[0]/r"] = 1.0;
        mag[path + "flux/position[0]/z"] = 0.0;
        mag[path + "flux/position[0]/phi"] = i * (M_PI / 5.0);
        mag[path + "flux/time"] = time;
        mag[path + "flux/data"] = data;
    }
```

```
    return 0;
}
```

C++ example

```
import pyhdc
import numpy as np
import math

tree = pyhdc.HDC()
tree["magnetics"] = None
mag = tree["magnetics"]

mag["ids_properties/homogeneous_time"] = 0

time = np.linspace(0.1, 10, 100)
data = np.ones(100) * 100.0

for i in range(10):
    path = "flux_loop[%d]" % i
    mag[path + "flux/position[0]/r"] = 1.0
    mag[path + "flux/position[0]/z"] = 0.0
    mag[path + "flux/position[0]/phi"] = i * (math.pi / 5.0)
    mag[path + "flux/time"] = time
    mag[path + "flux/data"] = data
```

Python example

```
program main
  use al_hdc
  use hdc_fortran
  implicit none

  integer :: status
  type(hdc_t) :: tree, mag
  real(kind=dp) :: time(100)
  real(kind=dp) :: data(100)
  character(len=100) :: path
  real, parameter :: Pi = 3.1415927
  integer :: i

  tree = hdc_new()
  mag = hdc_get_child(tree, "magnetics")

  call hdc_set(mag, "ids_properties/homogeneous_time", 0)

  time = (/(i, i=1,100,1)/) / 10.0
  data = 100.0
```

```
  do i = 1,10
    write (path, "(a,i1,a)") "flux_loop[", i, "]"
    call hdc_set(mag, path // "flux/position[0]/r", 1.0)
    call hdc_set(mag, path // "flux/position[0]/z", 0.0)
    call hdc_set(mag, path // "flux/position[0]/phi", i * (Pi / 5.0))
    call hdc_set(mag, path // "flux/time", time)
    call hdc_set(mag, path // "flux/data", data)
  end do
end program main
```

Fortran example

```
tree = HDC();
tree.set("magnetics", HDC());
mag = tree.get_child("magnetics");

mag.set("ids_properties/homogeneous_time", 0);

time = linspace(0.1, 10, 100);
data = ones([100]) * 100.0;

for i = 1:10
    path = sprintf("flux_loop[%d]", i);
    mag[strcat(path, "flux/position[0]/r"] = 1.0;
    mag[strcat(path, "flux/position[0]/z"] = 0.0;
    mag[strcat(path, "flux/position[0]/phi"] = i * (pi / 5.0);
    mag[strcat(path, "flux/time"] = time;
    mag[strcat(path, "flux/data"] = data;
end
```

Matlab example

```
import org.iter.imas.al_hdc.DataEntry;
import dev.libhdc.HDC;
import org.nd4j.linalg.api.ndarray.INDArray;

class Example
{
    public static void main(String args[]) {
        HDC tree = new HDC();
        HDC mag = tree["magnetics"];

        mag["ids_properties/homogeneous_time"] = 0;

        ArrayList<double> time(100);
        ArrayList<double> data(100);
        for (int i = 0; i < 100; i++) {
            time[i] = 0.1 * (i + 1);
```

```
            data[i] = 100.0;
        }

        for (int i = 0; i < 10; i++) {
            String path = "flux_loop[" + String.valueOf(i) + "]";

            mag.add_child(path + "flux/position[0]/r");
            mag.add_child(path + "flux/position[0]/z");
            mag.add_child(path + "flux/position[0]/phi");
            mag.add_child(path + "flux/time");
            mag.add_child(path + "flux/data");

            mag.get(path + "flux/position[0]/r").set_data(1.0);
            mag.get(path + "flux/position[0]/z").set_data(0.0);
            mag.get(path + "flux/position[0]/phi").set_data(i * (M_PI
/ 5.0));
            mag.get(path + "flux/time").set_data(time);
            mag.get(path + "flux/data").set_data(data);
        }
    }
}
```

Java example

5.17.6 Putting an entire IDS data

If you have obtained and modified an entire IDS structure using the HDC get or get_slice routines or you have created a new HDC tree containing an IDS structure you can put this whole structure into the IDS data entry using the put or put_slice methods. If you use the put_slice method the data contained in the tree should be the data for the time slice specified.

Because the top-level nodes of the tree are the IDS names (i.e., “magnetics”) then you do not need to provide a path to the put method.

For brevity, the examples below are shown putting an IDS that has been read from an existing IDS data entry you can generate the HDC tree by hand using the example shown in the previous section.

```
#include <al_hdc.h>

int main()
{
    imas::DataEntry entry
        {"imas:///iter?machine=test&shot=1000&run=0"};
    entry.open();

    imas::IDS equilibrium = entry.get("equilibrium");

    imas::DataEntry entry2
        {"imas:///iter?machine=test&shot=1000&run=1"};
```

```
    entry2.create();

    entry2.put("", equilibrium);

    entry.close();
    entry2.close();
    return 0;
}
```

C++ example

```
import al_hdc

entry = al_hdc.DataEntry("imas:///iter?machine=test&shot=1000&run=0")
entry.open()

equilibrium = file.get("equilibrium")

entry2 =
al_hdc.DataEntry("imas:///iter?machine=test&shot=1000&run=1")
entry2.create()

entry2.put("", equilibrium)

entry.close()
entry2.close()
```

Python example

```
program main
  use al_hdc
  implicit none

  integer :: idx, idx2
  integer :: status
  type(hdc_t) :: equilibrium

  idx = open_entry("imas:///iter?machine=test&shot=1000&run=0",
status)

  status = get_ids(idx, "equilibrium", equilibrium)

  idx2 = create_entry("imas:///iter?machine=test&shot=1000&run=1",
status)

  status = put_ids(idx2, "", equilibrium)

  call close_entry(idx, status)
  call close_entry(idx2, status)
```

```
end program main
```

Fortran example

```
entry = IMAS_HDC("imas:///iter?machine=test&shot=1000&run=0");
entry.open();

equilibrium = entry.get("equilibrium");

entry2 = IMAS_HDC("imas:///iter?machine=test&shot=1000&run=1");
entry2.create();

entry2.put("", equilibrium);

entry.close();
entry2.close();
```

Matlab example

```
import org.iter.imas.al_hdc.DataEntry;
import dev.libhdc.HDC;

class Example
{
    public static void main(String args[]) {
        DataEntry entry =
          new DataEntry("imas:///iter?machine=test&shot=1000&run=0");
        entry.open();

        HDC equilibrium = entry.get("equilibrium");

        DataEntry entry2 =
          new DataEntry("imas:///iter?machine=test&shot=1000&run=1");
        entry2.create();

        entry2.put("", equilibrium);

        entry.close();
        entry2.close();
    }
}
```

Java example

5.17.7 Putting partial IDS data

Instead of putting an entire IDS, it is possible to only put part of the IDS structure. You can do this by specified the path in the IDS structure of the root of the data tree being given.

If the IDS data entry is newly created, you will first need to write “ids_properties/homogeneous_time” for the IDS you are trying to write.

Partial put operations may create inconsistencies in the stored IDS (either physical inconsistencies or structural inconsistencies in the Data Dictionary sense, e.g. shape conflicts) and should be used very carefully. For example, if you PUT a node, make sure to PUT also its coordinates and to use consistent sizes.

```
#include <al_hdc.h>
#include <algorithm>

int main()
{
    imas::DataEntry entry
        {"imas:///iter?machine=test&shot=1000&run=2"};
    entry.create();

    // Need to put this first.
    imas::IDS ids;
    ids["magnetics/ids_properties/homogeneous_time"] = 0;
    entry.put("", ids);

    imas::IDS flux_loop_3;
    flux_loop_3["position/r"] = 1.0;
    flux_loop_3["position/z"] = 2.0;

    entry.put("magnetics/flux_loop[2]", flux_loop_3);

    entry.close();
    return 0;
}
```

C++ example

```
import al_hdc
import pyhdc

entry = al_hdc.DataEntry("imas:///iter?machine=test&shot=1000&run=2")
entry.create()

ids = pyhdc.HDC()
ids["magnetics/ids_properties/homogeneous_time"] = 0
entry.put("", ids)

flux_loop_3 = pyhdc.HDC()
flux_loop_3["position/r"] = 1.0
flux_loop_3["position/z"] = 2.0
```

```
entry.put("magnetics/flux_loop[2]", flux_loop_3)

entry.close()
```

Python example

```
program main
  use al_hdc
  use hdc_fortran
  implicit none

  integer :: idx
  integer :: status
  type(hdc_t) :: magnetics, flux_loop_3

  idx = create_entry("imas:///iter?machine=test&shot=1000&run=2", &
      status)

  magnetics = hdc_new()
  call hdc_set(magnetics, &
      "magnetics/ids_properties/homogeneous_time", 0)
  status = put_ids(idx, "", magnetics)

  flux_loop_3 = hdc_new()
  call hdc_set(flux_loop_3, "position/r", 1.0)
  call hdc_set(flux_loop_3, "position/z", 2.0)

  status = put_ids(idx, "magnetics/flux_loop[2]", flux_loop_3)

  call close_entry(idx, status)
end program main
```

Fortran example

```
entry = IMAS_HDC("imas:///iter?machine=test&shot=1000&run=2");
entry.create();

magnetics = HDC();
magnetics.set("magnetics/ids_properties/homogeneous_time", 0);

entry.put("", magnetics);

flux_loop_3 = HDC();
flux_loop_3.set("position/r", 1.0);
flux_loop_3.set("position/z", 2.0);

entry.put("magnetics/flux_loop[2]", flux_loop_3);

entry.close();
```

Matlab example

```
import org.iter.imas.al_hdc.DataEntry;
import dev.libhdc.HDC;

class Example
{
    public static void main(String args[]) {
        DataEntry entry =
          new DataEntry("imas:///iter?machine=test&shot=1000&run=2");
        entry.create();

        // Need to put this first.
        HDC ids = new HDC();
        ids["magnetics/ids_properties/homogeneous_time"] = 0;
        entry.put("", ids);

        HDC flux_loop_3 = new HDC();
        flux_loop_3["position/r"] = 1.0;
        flux_loop_3["position/z"] = 2.0;

        entry.put("magnetics/flux_loop[2]", flux_loop_3);

        entry.close();
    }
}
```

Java example

6 Tools for rapid browsing of the content of the IMAS data base folders and data entries

The `idstools` package provides script for rapid browsing of the content of the IMAS data base folders and data entries. To use it, first load the corresponding module:

```
$ module load idstools
```

6.1 Listing all data base entries for a given user

To list all data entries for the current user, type:

```
$ imasdbs
```

To list all data entries for a given `username`, type:

```
$ imasdbs -u username
```

To list all data entries for a given databasename, type:

```
$ imasdbs -t databasename
```

To list all data entries for a given IMASMajorVersion, type:

```
$ imasdbs -v IMASMajorVersion
```

All options above can be combined to do a finer selection in a single line.

6.2 Listing all IDSs in a given data entry

To list all IDSs contained in a given data entry, type:

```
$ listidss [-u username] [-t databasename] [-v IMASMajorVersion]
pulse,run
```

By default, username is the current user, IMASMajorVersion is defined by the currently loaded IMAS module, and databasename is defined by the latest call to the imasdb script (see 5.1).

pulse,run must be provided separated by a comma.

6.3 Listing the content of an IDS

To list the content (all data) of an IDS, type:

```
$ idsdump [username] [databasename] [IMASMajorVersion] pulse run
IDSOccurrence
```

IDSOccurrence is a string indicating the name and occurrence of the target IDS in the Data Entry (see 4.1)

By default, username is the current user, IMASMajorVersion is defined by the currently loaded IMAS module, and databasename is defined by the latest call to the imasdb script (see 5.1).

6.4 Retrieving data subset of an IDS

This is the Python script version of the partial GET described in 5.13.

To print only to a subset of IDS data, type:

```
$ idspartialget username databasename IMASMajorVersion pulse run
IDSOccurrence dataPath
```

IDSOccurrence is a string indicating the name and occurrence of the target IDS in the Data Entry (see 4.1)

username is the user login name,

IMASMajorVersion is defined by the currently loaded IMAS module,

databasename is defined by the latest call to the imasdb script

dataPath defines path to IDS node(s) (see 5.13).

6.5 Copying all or selected IDS from one data entry to another

The idscopy script copies a list of IDS from a data entry specified by (username, databasename IMASMajorVersion, pulse and run) to a new destination data entry created in your database set by imasdb, via a series of GET/PUT operations.

If the destination data entry was pre-existing, it's previous content is erased in the operation.

This script is somehow superseded by the more flexible imasmerge_entries script (see below).

Usage:

```
$ idscopy username databasename IMASMajorVersion pulse run
destination_pulse destination_run [IDS-list-as-Python-list]
```

Example, copy all IDSs:

```
idscopy coster jet 3 71827 21 71827 30
```

Example, copy just coreprof and equilibrium:

```
idscopy coster jet 3 71827 21 71827 30 "['core_profiles',
'equilibrium']"
```

6.6 Copying all or selected IDS from one data entry to another existing one

This imasmerge_entries script copies all valid IDSs (optionnally, you can specify a list of IDS names) from a given source IMAS data entry to a destination data entry. Unlike the previous script idscopy, it doesn't erase the destination entry and thus provides a form of merge functionality between IMAS data entries.

If a given IDS already exists in the destination data entry, it skips it by default or overwrites it if you have selected the -overwrite option. If the destination data entry doesn''t exist, it creates it. All IDS occurrences will be copied. If you provide a list of IDS names, all existing occurrences in the source will be copied to the destination. NOTE : the destination IMAS_MAJOR_VERSION is defined by the loaded IMAS module. The source IMAS_MAJOR_VERSION as well, unless you specify the -v option.

```
$  imasmerge_entries [-u source_user] [-v DD_major_version] [-
overwrite] source_machine source_pulse source_run destination_machine
```

```
destination_pulse destination_run [-ids_list 'IDS-list-as-Python-list']
```

Example, copy all IDSs:

```
imasmerge_entries jet 71827 21 jet 71827 30
```

Example, copy just core_profiles and equilibrium:

```
imasmerge_entries -u source_user -v 3 -overwrite jet 71827 21 jet 71827 30 -ids_list '[\'core_profiles\', \'equilibrium\']'
```

6.7 Finding different values between two IDSs of the same type

The `idsdiff` script scans all nodes of a given IDS in two given data entries and reports any difference in the node values. The data entries are looked for within the database set by `imasdb`.

Usage:

```
$ idsdiff IDSname pulse1 run1 pulse2 run2
```

NB: presently the script works only on the default (0) occurrence of IDSs in each data entry

Example :

```
idsdiff core_profiles 400 2 400 3
```

6.8 Finding non-empty occurrences of an IDS in a data entry

The `idsoccurrence` script reports non-empty occurrences of a specified IDS (search is currently limited to 'maxocc' occurrences, default to 15).

```
$ idsoccurrences [-h] [-u user_name] [-v version] [-c] [-maxocc maxocc] database shot run ids
```

Example for searching occurrences of 'equilibrium' IDS in shot 54178, run 0, of 'west' database for user 'imas':

```
idsoccurrences -u imas west 54178 0 equilibrium
```

6.9 Reading the DD version of an IMAS data entry (MDSplus backend)

The `printMDSplusFileVersion` script returns the DD version of an IMAS data file written using the MDSplus backend of the Access Layer. In the MDSplus backend, the structure of the file is frozen and corresponds to the DD version with which the file has been created. This limitation of frozen file structure may create incompatibility issues, e.g. when trying to PUT an IDS with a more recent version of the DD in an older file, if the structure of that IDS has changed. The solution to this particular problem is to create a

new pulse file with the desired DD version and use the `imasmerge_entries` script to copy IDSs from older pulse files into the new one.

The script will also return the AL version with which the file was created, although this information is less critical than the DD version (for the reason explained above).

```
$ printMDSplusFileVersion [-u user] [-v version] database shot run
```

NB : this script is part of the Access Layer directly, not of IDStools, since it is related to the MDSbackend.