XIOS reference guide

Yann Meurdesoif

October 10, 2014

Chapter 1

Attribute reference

1.1 Context attribute reference

calendard_type : *enum { Gregorian, Julian, D360, AllLeap, NoLeap}*

Fortran :

```
CHARACTER(LEN=*) :: calendard_type
```

Define the calendar used for the context. This attribute is mandatory

start_date : *date*

Fortran :

```
CHARACTER(LEN=*) :: start_date
```

Define the start date of the simulation for the current context. This attribute is madatory

time_origin : *date*

Fortran :

```
CHARACTER(LEN=*) :: time_origin
```

Define the time origin of the time axis. It will appear as metadata attached to the time axis in the output file. The default value used the **start_date** attribute.

timestep : *duration*

Fortran :

```
CHARACTER(LEN=*) :: timestep
```

Define the timestep of the context. This attribute is mandatory. The time step can be also defined throw the fortran subroutine :

```
SUBROUTINE xios_set_timestep(timestep)
TYPE(xios_time) :: timestep
```

1.2 Axis attribute reference

name : *string*

Fortran :

```
CHARACTER(LEN=*) :: name
```

Define the name of the vertical axis, as it will appear in a file. If not defined, a name is self generated from the id. If multiple vertical axis are defined in a same file, each name must be different.

standard_name : *string*

Fortran :

```
CHARACTER(LEN=*) :: standard_name
```

Define the standard name of the vertical axis, as it will appear in the metadata attached to the axis of the output file.

long_name : *string*

Fortran :

```
CHARACTER(LEN=*) :: long_name
```

Define the long name of the vertical axis, as it will appear in the metadata attached to the axis of the output file.

unit : *string*

Fortran :

```
CHARACTER(LEN=*) :: unit
```

Define the unit of the axis as it will appear in the metadata attached to the axis in the output file.

size : *integer*

Fortran :

```
INTEGER :: size
```

Define the size of the axis. This attribute is mandatory.

zoom_begin : *integer*

Fortran :

```
INTEGER :: zoom_begin
```

Define the the begining of the zoom. This must be an index betwen 1 and **size**. If not specified, default value is 1. It must also be evaluated from **zoom_end** and **zoom_size** if there are specified.

zoom_end : *integer*

Fortran :

```
INTEGER :: zoom_end
```

Define the the end of the zoom. This must be an index betwen 1 and **size**. If not specified, default value is **size**. It must also be evaluated from **zoom_begin** and **zoom_size** if there are specified.

zoom_size : *integer*

Fortran :

```
INTEGER :: zoom_size
```

Define the the size of the zoom. This must be an integer betwen 1 and **size**. If not specified, default value is **size**. It must also be evaluated from **zoom_begin** and **zoom_end** if there are specified.

value : *1D-array of double*

Fortran :

```
DOUBLE PRECISION :: value(:)
```

Define the level values of the vertical axis. The size of the array must be equal to the **size** attribute. If not defined default value are filled with value from 1 to **size**.

1.3 Domain attribute reference

name : *string*

Fortran :

```
CHARACTER(LEN=*) :: name
```

Define the name of the horizontal domain. This attribute may be used in case of multiple domain defined in a same file. In this case, the **name** attribute will be suffixed to the longitude and latitude dimensions and axis name. Otherwise a suffix will be self-generated.

ni_glo : *integer*

Fortran :

```
INTEGER :: ni_glo
```

Define the first dimension of the global domain. This attribute is mandatory.

nj_glo : *integer*

Fortran :

```
INTEGER :: nj_glo
```

Define the second dimension of the global domain. This attribute is mandatory.

ni : *integer*

Fortran :

```
INTEGER :: ni
```

Define the first dimension of the local domain. This attribute may be also computed from **ibegin** and **iend** attribute value, so this attribute is optional but a compliant value must be fixed.

$ni = iend - ibegin + 1$

$1 \leq ibegin \leq iend \leq ni\_glo$

ibegin : *integer*

Fortran :

```
INTEGER :: ibegin
```

Define the begining index of the first dimension of the local domain. This attribute may be also computed from **ni** and **iend** attribute value, so this attribute is optional but a compliant value must be fixed.

$ibegin = iend - ni + 1$

$1 \leq ibegin \leq iend \leq ni\_glo$

iend : *integer*

Fortran :

```
INTEGER :: iend
```

Define the ending index of the first dimension of the local domain. This attribute may be also computed from **ni** and **ibegin** attribute value, so this attribute is optional but a compliant value must be fixed.

$iend = ibegin + ni - 1$

$1 \leq ibegin \leq iend \leq ni\_glo$

nj : *integer*

Fortran :

```
INTEGER :: nj
```

Define the second dimension of the local domain. This attribute may be also computed from **jbegin** and **jend** attribute value, so this attribute is optional but a compliant value must be fixed.

$nj = jend - jbegin + 1$

$1 \leq jbegin \leq jend \leq nj\_glo$

jbegin : *integer*

Fortran :

```
INTEGER :: jbegin
```

Define the begining index of the second dimension of the local domain. This attribute may be also computed from **nj** and **jend** attribute value, so this attribute is optional but a compliant value must be fixed.

$jbegin = jend - nj + 1$

$1 \leq jbegin \leq jend \leq nj\_glo$

jend : *integer*

Fortran :

```
INTEGER :: jend
```

Define the ending index of the second dimension of the local domain. This attribute may be also computed from **nj** and **jbegin** attribute value, so this attribute is optional but a compliant value must be fixed.

$jend = jbegin + nj - 1$

$1 \leq jbegin \leq jend \leq nj\_glo$

zoom_ni : *integer*

Fortran :

```
INTEGER :: zoom_ni
```

Define the size of the zoom on the first dimension on the global domain. This attribute is optionnal and take as default value the value of **ni**.

$1 \leq zoom\_ni \leq ni$

zoom_ibegin : *integer*

Fortran

```
INTEGER :: zoom_ibegin
```

Define the begining index on the first dimension of the zoom for the global domain. This attribute is optionnal and take **1** as default value.

zoom_nj : *integer*

Fortran :

```
INTEGER :: zoom_nj
```

Define the size of the zoom on the second dimension for the global domain. This attribute is optionnal and take as default value the value of **nj**.

$1 \leq zoom\_ni \leq ni$

zoom_jbegin : *integer*

Fortran :

```
INTEGER :: zoom_jbegin
```

Define the begining index on the sencond dimension of the zoom on the global domain. This attribute is optionnal and take **1** as default value.

mask : *2D-array of bool*

Fortran :

```
LOGICAL :: mask(:,:)
```

Define the mask of the local domain. Masked value will be replaced by the value of the field attribute **default_value** in the output file.

lonvalue : *1D-array of double*

Fortran :

```
DOUBLE PRECISION :: lonvalue(:)
```

Define the value of the longitude on the local domain. In case of cartesian grid, the size of the array will be **ni**. In case of curvilinear grid, the size of the array will be **ni**×**nj**. The attribute is mandatory.

latvalue : *1D-array of double*

Fortran :

```
DOUBLE PRECISION :: latvalue(:)
```

Define the value of the latitude on the local domain. In case of cartesian grid, the size of the array will be nj. In case of curvilinear grid, the size of the array will be **ni**×**nj**. The attribute is mandatory.

data_dim : *integer*

Fortran :

```
INTEGER :: datadim
```

Define how a field is store on memory for the client code. **datadim** value can be **1** or **2**. A value of **1** indicate that the horizontal layer of the field is stored on a 1D array as a vector of point. A value of **2** indicate that the horizontal layer is stored in a 2D array. This attribute is mandatory.

data_ibegin : *integer*

Fortran :

```
INTEGER :: data_ibegin
```

Define the begining index of the field data for the first dimension. This attribute is an offset regarding the local domain, so the value can be negative. A negative value indicate that only some valid part of the data will extracted, for example in case of ghost cell. A positive value indicate that the local domain is greater than the data stored in memory. A 0-value means that the local domain match the data in memory. This attribute is optionnal and the default value is 0. Otherwise **data_ibegin** and **data_ni** must be defined together.

data_ni : *integer*

Fortran :

```
INTEGER :: data_ni
```

Define the size of the field data for the first dimension. This attribute is optionnal and the default value is **ni**. Otherwise **data_ibegin** and **data_ni** must be defined together.

data_jbegin : *integer*

Fortran :

```
INTEGER :: data_jbegin
```

Define the begining index of the field data for the second dimension. This attribute is take account only if **data_dim=2**. This attribute is an offset regarding the local domain, so the value can be negative. A negative value indicate that only some valid part of the data will extracted, for example in case of ghost cell. A positive value indicate that the local domain is greater than the data stored in memory. A 0-value means that the local domain match the data in memory. This attribute is optionnal and the default value is **0**. Otherwise **data_jbegin** and **data_nj** must be defined together.

data_nj : *integer*

Fortran :

```
INTEGER :: data_nj
```

Define the size of the field data for the first dimension. This attribute is take account only if **data_dim=2**. This attribute is optionnal and the default value is **nj**. Otherwise **data_jbegin** and **data_nj** must be defined together.

data_n_index : *integer*

Fortran :

```
INTEGER :: data_nindex
```

In case of a compressed horizontal domain, this attribute define the number of points stored in memory on the local domain.

data_i_index : *1D-array of integer*

Fortran :

```
INTEGER :: data_i_index(:)
```

In case of a compressed horizontal domain, define the indexation the indexation of the data for the first dimension. The size of the array must be **data_nindex**. This attribute is optionnal.

data_j_index : *1D-array of integer*

Fortran :

```
INTEGER :: data_j_index(:)
```

In case of a compressed horizontal domain, define the indexation the indexation of the data for the second dimension. This is meaningful only if **data_dim=2**. The size of the array must be **data_nindex**. This attribute is optionnal.

1.4 Grid attribute reference

name : string

Fortran :

```
CHARACTER(LEN=*) :: name
```

Define the name of the grid. This attribute is actually not used internally. Optionnal attribute.

domain_ref : string

Fortran :

```
CHARACTER(LEN=*) :: domain_ref
```

Define the horizontal domain reference of the grid. This attribute is mandatory.

axis_ref : string

Fortran :

```
CHARACTER(LEN=*) :: axis_ref
```

Define the axis reference of the grid. This attribute is optionnal, if not defined, the grid will be considered as a 2-Dimensionnal grid without vertical layer.

1.5 Field attribute reference

name : *string*

Fortran :

```
CHARACTER(LEN=*) :: name
```

Define the **name** of the field as it will apear in an ouput file. This attribut is optionnal. If not present, the identifer **id** will be substituted.

standard_name : *string*

Fortran :

```
CHARACTER(LEN=*) :: standard_name
```

Define the **standard_name** attribute as it will appear in the metadata of an ouput file. This attribute is optionnal.

long_name : *string*

Fortran :

```
CHARACTER(LEN=*) :: long_name
```

Define the **long_name** attribute as it will appear in the metadata of an output file. This attribute is optionnal.

unit : *string*

Fortran :

```
CHARACTER(LEN=*) :: unit
```

Define the **unit** of the field. This attribute is optionnal.

operation : *enum { once, instant, average, maximum, minimum, accumulate }*

Fortran :

```
CHARACTER(LEN=*) :: operation
```

Define the temporal operation applied on the field. This attribute is mandatory.

freq_op : *duration*

Fortran :

```
CHARACTER(LEN=*) :: freq_op
```

Define the frequency of the sampling for the temporal operation, so a field value will be used for temporal averaging every **freq_op** timestep. It is very usefull for sub-processus called at different frequency in a model. This attribute is optionnal, the default value is **1ts**(1 time step).

freq_offset : *duration*

Fortran :

```
CHARACTER(LEN=*) :: freq_offset
```

Define the offset when **freq_op** is defined. This attribute is optionnal, the default value is **0ts**(0 time step).
$0 \leq freq\_offset < freq\_op$

level : *integer*

Fortran :

```
INTEGER :: level
```

Define the level of output of the field. A field will be output only if the file attribute **output_level** $\geq$**level**. This attribute is optionnal, the default value is **0**.

prec : *integer*

Fortran :

```
INTEGER :: prec
```

Define the precision in byte of a field in an output file. For now only value **4** and **8** are correct. This attribute is optionnal, the default value is **8**.

enabled : *bool*

Fortran :

```
LOGICAL :: enabled
```

Define if a field must be output or not. This attribut is optionnal, the default value is **true**.

field_ref : *string*

Fortran :

```
CHARACTER(LEN=*) :: field_ref
```

Define a field reference. All attributes are inherited from the referenced field after the classical inheritence mechanism. The value assigned to the referenced field is transmitted to to current field to perform temporal operation. This attribute is optionnal.

grid_ref : *string*

Fortran :

```
CHARACTER(LEN=*) :: grid_ref
```

Define on which grid the current field is defined. This attribut is optionnal, if missing, domain_ref and axis_ref must be defining.

domain_ref : *string*

Fortran :

```
CHARACTER(LEN=*) :: domain_ref
```

Define on which horizontal domain the current field is defined. This attribut is optionnal, but if this attribute is defined, **grid_ref** must not be.

axis_ref : *string*

Fortran :

```
CHARACTER(LEN=*) :: axis_ref
```

Define on which vertical axis the current field is defined. This attribute is optionnal, but if this attribute is defined, **domain_ref** must be too and **grid_ref** must not.

default_value : *double*

Fortran :

```
DOUBLE PRECISION :: default_value
```

Define the value for the missing data of a field. This attribute is optionnal. The default value is **0**.

1.6 File attribute reference

name : *string*

Fortran :

```
CHARACTER(LEN=*) :: name
```

Define the name of the ouput file. This attribute is mandatory.

name_suffix : *string*

Fortran :

```
CHARACTER(LEN=*) :: name_suffix
```

Define a suffix to add to the name of the output file. Thisn attribute is optionnal.

output_freq : *duration*

Fortran :

```
CHARACTER(LEN=*) :: output_freq
```

Define the output frequency for the current file. This attribute is mandatory.

output_level : *integer*

Fortran :

```
INTEGER :: output_level
```

Define an output level for the field defining inside the current file. Field is output only if the field attribute $level \leq output\_level$.

sync_freq : *duration*

Fortran :

```
CHARACTER(LEN=*) :: sync_freq
```

Define the frequency for flushing the current file onto disk. It may result bad performance but data are wrote even if the file will not be closed. This attribute is optionnal.

split_freq : *duration*

Fortran :

```
CHARACTER(LEN=*) :: split_freq
```

Define the time frequency for splitting the current file. Date is suffixed to the name (see *split_freq_format* attribute). This attribute is optionnal.

split_freq_format : *string*

Fortran :

```
CHARACTER(LEN=*) :: split_freq_format
```

Define the format of the split date suffixed to the file. Can contain any character, %y will be replaced by the year (4 characters), %mo month (2 char), %d day (2 char), %mi minute (2 char), %s second (2 char). This attribute is optionnal. and the default behavior is to create a suffix with the date until the smaller non zero unit. For example, in one day split frequency, the hour minute and second will note appear in the suffix, only year, month and day.

enabled : *bool*

Fortran :

```
LOGICAL :: enabled
```

Define if a file must be output or not. This attribute is mandatory, the default value is **true**.

type : *enumeration { one_file, multiple_file}*

Fortran :

```
CHARACTER(LEN=*) :: type
```

Define which type of file will be output. **multiple_file** : one file by server using sequential netcdf writing. **one_file** : one single global file is wrote using netcdf4 parallel access.

par_access : *enumeration { collective, independent }*

Fortran :

```
CHARACTER(LEN=*) :: par_access
```

For parallel writting, define the MPI call used. This attribut is optionnal, the default value is **collective**.

min_digits : *integer*

Fortran :

```
INTEGER :: min_digits
```

For multiple_file, define the minimum digits composing the suffix defining the rank of the server, which will be happened to the name of the output file. This attribut is optionnal and the default value is **0**.

Chapter 2

Fortran interface reference

Initialization

XIOS initialization

Synopsis :

```
SUBROUTINE xios_initialize(client_id, local_comm, return_comm)
  CHARACTER(LEN=*),INTENT(IN)         :: client_id
  INTEGER,INTENT(IN),OPTIONAL         :: local_comm
  INTEGER,INTENT(OUT),OPTIONAL        :: return_comm
```

Argument :

- `client_id` : client identifier
- `local_comm` : MPI communicator of the client
- `return_comm` : split return MPI communicator

Description :

This subroutine must be called before any other call of MPI client library. It may be able to initialize MPI library (calling `MPI_Init`) if not already initialized. Since XIOS is able to work in client/server mode (parameter `using_server=true`), the global communicator must be split and a local split communicator is returned to be used by the client model for it own purpose. If more than one model is present, XIOS could be interfaced with the OASIS coupler (compiled with `-using_oasis` option and parameter `using_oasis=true`), so in this case, the splitting would be done globally by OASIS.

- If MPI is not initialized, XIOS would initialize it calling MPI_Init function. In this case, the MPI finalization would be done by XIOS in the `xios_finalize` subroutine, and must not be done by the model.
- If OASIS coupler is not used (using_oasis=false)

- If server mode is not activated (`using_server=false`) : if local_comm MPI communicator is specified then it would be used for internal MPI communication otherwise `MPI_COMM_WORLD` communicator would be used by default. A copy of the communicator (of `local_comm` or `MPI_COMM_WORLD`) would be returned in return_comm argument. If `return_comm` is not specified, then `local_comm` or `MPI_COMM_WORLD` can be used by the model for it own communication.
- If server mode is activated (`using_server=true`) : `local_comm` must not be specified since the global `MPI_COMM_WORLD` communicator would be splitted by XIOS. The split communicator is returned in `return_comm` argument.

• If OASIS coupler is used (`using_oasis=true`)

- If server mode is not enabled (`using_server=false`)
  - If `local_comm` is specified, it means that OASIS has been initialized by the model and global communicator has been already split previously by OASIS, and passed as `local_comm` argument. The returned communicator would be a duplicate copy of `local_comm`.
  - Otherwise : if MPI was not initialized, OASIS will be initialized calling `prism_init_comp_proto` subroutine. In this case, XIOS will call `prism_terminate_proto` when `xios_finalized` is called. The split communicator is returned in `return_comm` argument using `prism_get_localcomm_proto` return argument.
- If server mode is enabled (`using_server=true`)
  - If `local_comm` is specified, it means that OASIS has been initialized by the model and global communicator has been already split previously by OASIS, and passed as local_comm argument. The returned communicator return_comm would be a split communicator given by OASIS.
  - Otherwise : if MPI was not initialized, OASIS will be initialized calling `prism_init_comp_proto` subroutine. In this case, XIOS will call `prism_terminate_proto` when `xios_finalized` is called. The split communicator is returned in `return_comm` argument using `prism_get_localcomm_proto` return argument.

Finalization

XIOS finalization

Synopsis :

```
SUBROUTINE xios_finalize()
```

Arguments :

None

Description :

This call must be done at the end of the simulation for a succesfull execution. It gives the end signal to the xios server pools to finish it execution. If MPI has been initialize by XIOS the MPI_Finalize will be called. If OASIS coupler has been initialized by XIOS, then finalization will be done calling `prism_terminate_proto` subroutine.

Tree elements management subroutines

This set of subroutines enable the models to interact, complete or query the XML tree data base. New elements or group of elements can be added as child in the tree, attributes of the elements can be set or query. The type of element actually avalaible are : context, axis, domain, grid, field, variable and file. An element can be identified by a string or by an handle associated to the type of the element. Root element (ex : "axis_definition", "field_definition",....) are considered like a group of element and are identified by a specific string "element_definition" where element can be any one of the existing elements.

Fortran type of the handles element

TYPE(txios_element)

where "element" can be any one among "context", "axis", "domain", "grid", "field", "variable" or "file", or the associated group (excepted for context) : "axis_group", "domain_group", "grid_group", "field_group", "variable_group" or "file_group".

Getting handles

Synopsis :

```
SUBROUTINE xios_get_element_handle(id,handle)
CHARACTER(len = *) , INTENT(IN) :: id
TYPE(txios_element), INTENT(OUT):: handle
```

whera element is one of the existing element or group of element.

Arguments :

- `id` : string identifier.
- `handle` : element handle

Description :

This subroutine return the handle of the specified element identified by its string. The element must be existing otherwise it raise an error.

Query for a valid element

Synopsis :

```
LOGICAL FUNCTION xios_is_valid_element(id)
CHARACTER(len = *) , INTENT(IN) :: id
```

whera element is one of the existing element or group of element.

Arguments :

- `id` : string identifier.

Description :

This function return .TRUE. if the element defined by the string identifier id is existing in the data base, otherwise it return .FALSE. .

Adding child

Synopsis :

```
SUBROUTINE xios_add_element(parent_handle, child_handle, child_id)
TYPE(txios_element)         , INTENT(IN) :: parent_handle
TYPE(txios_element)         , INTENT(OUT):: child_handle
CHARACTER(len = *), OPTIONAL, INTENT(IN) :: child_id
```

where element is one of the existing element or group of element.

Arguments :

- `parent_handle` : handle of the parent element.
- `child_handle` : handle of the child element.
- `child_id` : string identifier of the child.

Description :

This subroutine add a child to an existing parent element. The identifier of the child, if existing, can be specified optionnaly. All group elements can contains child of the same kind, provided generic inheritance. Some elements can contains childs of an other kind for a specific behaviour. File element may contains field_group, field, variable and variable_group child elements. Field elements may contains variable_group of variable child element.

Query if a value of an element attributes is defined (by handle)

Synopsis :

```
SUBROUTINE xios_is_defined_attr(handle, attr_1=attribute_1, attr_2=attribute_2, ...)
```

```
TYPE(txios_element)         , INTENT(IN) :: handle
LOGICAL, OPTIONAL  , INTENT(IN) :: attr_1
LOGICAL, OPTIONAL  , INTENT(IN) :: attr_2
....
```

where element is one of the existing element or group of element. attribute_x is describing in the chapter dedicated to the attribute description.

Arguments :

- `handle` : element handle.
- `attr_x` : return true if the attribute as a defined value.

Description :

This subroutine my be used to query if one or more attributes of an element have a defined value. The list of attributes and their type are described in a specific chapter of the documention.

Query if a value of an element attributes is defined (by identifier)

Synopsis :

```
SUBROUTINE xios_is_defined_element_attr(id, attr_1=attribute_1, attr_2=attribute_2, .
CHARACTER(len = *) , INTENT(IN) :: id
LOGICAL, OPTIONAL  , INTENT(IN) :: attr_1
LOGICAL, OPTIONAL  , INTENT(IN) :: attr_2
....
```

where element is one of the existing element or group of element. attribute_x is describing in the chapter dedicated to the attribute description.

Arguments :

- `id` : element identifier.
- `attr_x` : return true if the attribute as a defined value.

Description :

This subroutine my be used to query if one or more attributes of an element have a defined value. The list of available attributes and their type are described in a specific chapter of the documention.

Setting element attributes value by handle

Synopsis :

```
SUBROUTINE xios_set_attr(handle, attr_1=attribute_1, attr_2=attribute_2, ...)
```

```
TYPE(txios_element)         , INTENT(IN) :: handle
attribute_type_1, OPTIONAL  , INTENT(IN) :: attr_1
attribute_type_2, OPTIONAL  , INTENT(IN) :: attr_2
....
```

where element is one of the existing element or group of element. attribute_x and attribute_type_x are describing in the chapter dedicated to the attribute description.

Arguments :

- `handle` : element handle.
- `attr_x` : value of the attribute to be set.

Description :

This subroutine my be used to set one or more attribute to an element defined by its handle. The list of available attributes and their type are described in a specific chapter of the documention.

Setting element attributes value by id

Synopsis :

```
SUBROUTINE xios_set_element_attr(id, attr_1=attribute_1, attr_2=attribute_2, ...)
CHARACTER(len = *),  INTENT(IN)          :: id
attribute_type_1, OPTIONAL  , INTENT(IN) :: attr_1
attribute_type_2, OPTIONAL  , INTENT(IN) :: attr_2
....
```

where element is one of the existing element or group of element. attribute_x and attribute_type_x are describing in the chapter dedicated to the attribute description.

Arguments :

- `id` : string identifier.
- `attr_x` : value of the attribute to be set.

Description :

This subroutine my be used to set one or more attribute to an element defined by its string id. The list of available attributes and their type are described in a specific chapter of the documention.

Getting element attributes value (by handle)

Synopsis :

```
SUBROUTINE xios_get_attr(handle, attr_1=attribute_1, attr_2=attribute_2, ...)
TYPE(txios_element)          , INTENT(IN) :: handle
attribute_type_1, OPTIONAL   , INTENT(OUT) :: attr_1
attribute_type_2, OPTIONAL   , INTENT(OUT) :: attr_2
....
```

where element is one of the existing element or group of element. attribute_x and attribute_type_x are describing in the chapter dedicated to the attribute description.

Arguments :

- `handle` : element handle.
- `attr_x` : value of the attribute to be get.

Description :

This subroutine my be used to get one or more attribute value of an element defined by its handle. All attributes in the arguments list must be defined. The list of available attributes and their type are described in a specific chapter of the documention.

Getting element attributes value (by identifier)

Synopsis :

```
SUBROUTINE xios_get_element_attr(id, attr_1=attribute_1, attr_2=attribute_2, ...)
CHARACTER(len = *),   INTENT(IN)          :: id
attribute_type_1, OPTIONAL   , INTENT(OUT) :: attr_1
attribute_type_2, OPTIONAL   , INTENT(OUT) :: attr_2
....
```

where element is one of the existing element or group of element. attribute_x is describing in the chapter dedicated to the attribute description.

Arguments :

- `id` : element string identifier.
- `attr_x` : value of the attribute to be get.

Description :

This subroutine my be used to get one or more attribute value of an element defined by its handle. All attributes in the arguments list must have a defined value. The list of available attributes and their type are described in a specific chapter of the documention.

Interface relative to context managment

XIOS context initialization

Synopsis :

```
SUBROUTINE xios_context_initialize(context_id, context_comm)
  CHARACTER(LEN=*),INTENT(IN)          :: context_id
  INTEGER,INTENT(IN)                   :: context_comm
```

Argument :

- `context_id` : context identifier
- `context_comm` : MPI communicator of the context

Description :

This subroutine initialize a context identified by `context_id` string and must be called before any call related to this context. A context must be associated to a communicator, which can be the returned communicator of the `xios_initialize` subroutine or a sub-communicator of this. The context initialization is dynamic and can be done at any time before the `xios_finalize` call.

XIOS context finalization

Synopsis :

```
SUBROUTINE xios_context_finalize()
```

Arguments :

None

Description :

This subroutine must be call to close a context, before the `xios_finalize` call. It waits until that all pending request sent to the servers will be processed and the opened files will be closed.

Setting current active context

Synopsis :

```
SUBROUTINE xios_set_current_context(context_handle)
TYPE(txios_context),INTENT(IN) :: context_handle
```

or

```
SUBROUTINE xios_set_current_context(context_id)
CHARACTER(LEN=*),INTENT(IN) :: context_id
```

Arguments :

- `context_handle` : handle of the context

or

- `context_id` : string context identifier

Description :

These subroutines set the current active context. All xios calls after will refer to this active context. If only one context is defined, it is automatically set as the active context.

Closing definition

Synopsis :

```
SUBROUTINE xios_close_context_definition()
```

Arguments :

None

Description :

This subroutine must be call when all definitions of a context is finished at the end of the initialization and before entering to the time loop. A lot of operations are performed internally (inheritance, grid definition, contacting servers,...) so this call is mandatory. Any call related to the tree management definition done after will have an indefined effect.

Defining the time step of the context

Synopsis :

```
SUBROUTINE xios_set_timestep(timestep)
TYPE(txios_time),INTENT(IN) :: timestep
```

Arguments :

- `timestep` : timestep of the model using time units

Description :

This subroutine fix the timestep of the current context. Must be call before closing context definition.