XIOS Fortran Reference Guide

June 25, 2020

Chapter 1

Attribute reference

1.1 Context attribute reference

1.2 Calendar attribute reference

type: *enumeration {Gregorian, Julian, D360, AllLeap, NoLeap, user_defined}*

Fortran:

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

Define the calendar used for the current context. This attribute is mandatory and cannot be modified once it has been set.

When using the Fortran interface, this attribute must be defined using the following subroutine:

```
SUBROUTINE xios_define_calendar(type, timestep, start_date, time_origin,
                                day_length, month_lengths, year_length,
                                leap_year_month, leap_year_drift,
                                leap_year_drift_offset)
```

start_date: *date*

Fortran:

```
TYPE(xios_date) :: start_date
```

Define the start date of the simulation for the current context. This attribute is optional, the default value is *0000-01-01 00:00:00*. The **type** attribute must always be set at the same time or before this attribute is defined.

A partial date is allowed in the configuration file as long as the omitted parts are at the end, in which case they are initialized as in the default value. Optionally an offset can be added to the date using the notation "+ *duration*".

When using the Fortran interface, this attribute can be defined at the same time as the calendar **type**:

```
SUBROUTINE xios_define_calendar(type, timestep, start_date, time_origin,
                                day_length, month_lengths, year_length,
                                leap_year_month, leap_year_drift,
                                leap_year_drift_offset)
```

or later using the following subroutine:

```
SUBROUTINE xios_set_start_date(start_date)
```

time_origin: *date*

Fortran:

```
TYPE(xios_date) :: time_origin
```

Define the time origin of a time axis. It will appear as metadata attached to the time axis in an output file. This attribute is optional, the default value is ***0000-01-01 00:00:00***. The **type** attribute must always be set at the same time or before this attribute is defined.

A partial date is allowed in the configuration file as long as the omitted parts are at the end, in which case they are initialized as in the default value. Optionally an offset can be added to the date using the notation "+ *duration*".

When using the Fortran interface, this attribute can be defined at the same time as the calendar **type**:

```
SUBROUTINE xios_define_calendar(type, timestep, start_date, time_origin,
                                day_length, month_lengths, year_length,
                                leap_year_month, leap_year_drift,
                                leap_year_drift_offset)
```

or later using the following subroutine:

```
SUBROUTINE xios_set_time_origin(time_origin)
```

timestep: *duration*

Fortran:

```
TYPE(xios_duration) :: timestep
```

Define the time step of the simulation for the current context. This attribute is mandatory.

When using the Fortran interface, this attribute can be defined at the same time as the calendar **type**:

```
SUBROUTINE xios_define_calendar(type, timestep, start_date, time_origin,
                                day_length, month_lengths, year_length,
                                leap_year_month, leap_year_drift,
                                leap_year_drift_offset)
```

or using the following subroutine:

```
SUBROUTINE xios_set_timestep(timestep)
```

day_length: *integer*

Fortran:

```
INTEGER :: day_length
```

Define the duration of a day in seconds when using a custom calendar. This attribute is mandatory if the calendar **type** is set to "*user_defined*", otherwise it must not be defined.

When using the Fortran interface, this attribute must be defined at the same time as the calendar **type**:

```
SUBROUTINE xios_define_calendar(type, timestep, start_date, time_origin,
                                day_length, month_lengths, year_length,
                                leap_year_month, leap_year_drift,
                                leap_year_drift_offset)
```

month_lengths: *1D-array of integer*

Fortran:

```
INTEGER :: month_lengths(:)
```

Define the duration of each month, in days, when using a custom calendar. The number of elements in the array defines the number of months in a year and the sum of all elements is the total number of days in a year. This attribute is mandatory if the calendar **type** is set to ***user_defined*** and the **year_length** attribute is not used, otherwise it must not be defined.

When using the Fortran interface, this attribute must be defined at the same time as the calendar **type**:

```
SUBROUTINE xios_define_calendar(type, timestep, start_date, time_origin,
                                day_length, month_lengths, year_length,
                                leap_year_month, leap_year_drift,
                                leap_year_drift_offset)
```

year_length: *integer*

Fortran:

```
INTEGER :: year_length
```

Define the duration of a year, in seconds, when using a custom calendar. This attribute is mandatory if the calendar **type** is set to ***user_defined*** and the **month_lengths** attribute is not used, otherwise it must not be defined.

Note that the date format is modified when using this attribute: the month must be always be omitted and the day must also be omitted if $year\_length \leq day\_length$.

When using the Fortran interface, this attribute must be defined at the same time as the calendar **type**:

```
SUBROUTINE xios_define_calendar(type, timestep, start_date, time_origin,
                                day_length, month_lengths, year_length,
                                leap_year_month, leap_year_drift,
                                leap_year_drift_offset)
```

leap_year_month: *integer*

Fortran:

```
INTEGER :: leap_year_month
```

Define the month to which the extra day will be added in case of leap year, when using a custom calendar. This attribute is optional if the calendar **type** is set to ***user_defined*** and the **month_lengths** attribute is used, otherwise it must not be defined. The default behavior is not to have any leap year. If defined, this attribute must comply with the following constraint: $1 \leq leap\_year\_month \leq size(month\_lengths)$ and the **leap_year_drift** attribute must also be defined.

When using the Fortran interface, this attribute must be defined at the same time as the calendar **type**:

```
SUBROUTINE xios_define_calendar(type, timestep, start_date, time_origin,
                                day_length, month_lengths, year_length,
                                leap_year_month, leap_year_drift,
                                leap_year_drift_offset)
```

leap_year_drift: *double*

Fortran:

```
DOUBLE PRECISION :: leap_year_drift
```

Define the yearly drift, expressed as a fraction of a day, between the calendar year and the astronomical year, when using a custom calendar. This attribute is optional if the calendar **type** is set to ***user_defined*** and the **month_lengths** attribute is used, otherwise it must not be defined. The default behavior is not to have any leap year, i.e. the default value is **0**. If defined, this attribute must comply with the following constraint: $0 \leq leap\_year\_drift < 1$ and the **leap_year_month** attribute must also be defined.

When using the Fortran interface, this attribute must be defined at the same time as the calendar **type**:

```
SUBROUTINE xios_define_calendar(type, timestep, start_date, time_origin,
                                day_length, month_lengths, year_length,
                                leap_year_month, leap_year_drift,
                                leap_year_drift_offset)
```

leap_year_drift_offset: *double*

Fortran:

```
DOUBLE PRECISION :: leap_year_drift_offset
```

Define the initial drift between the calendar year and the astronomical year, expressed as a fraction of a day, at the beginning of the time origin's year, when using a custom calendar. This attribute is optional if the **leap_year_month** and **leap_year_drift** attributes are used, otherwise it must not be defined. The default value is **0**. If defined, this attribute must comply with the following constraint: $0 \leq leap\_year\_drift\_offset < 1$. If $leap\_yeap\_drift\_offset + leap\_yeap\_drift$ is greater or equal to 1, then the first year will be a leap year.

When using the Fortran interface, this attribute must be defined at the same time as the calendar **type**:

```
SUBROUTINE xios_define_calendar(type, timestep, start_date, time_origin,
                                day_length, month_lengths, year_length,
                                leap_year_month, leap_year_drift,
                                leap_year_drift_offset)
```

1.3 Scalar attribute reference

name (optional): *string*

Fortran:

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

Defines the name of a scalar as it will appear in a file. If not defined, the name will be generated automatically based on the id. If multiple scalars are defined in the same file, each scalar must have a unique name.

standard_name (optional): *string*

Fortran:

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

Defines the standard name of a scalar as it will appear in the scalar's metadata in an output file.

long_name (optional): *string*

Fortran:

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

Defines the long name of a scalar as it will appear in the scalar's metadata in an output file.

unit (optional): *string*

Fortran:

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

Defines the scalar unit as it will appear in the scalar's metadata in an output file.

value (optional): *double*

Fortran:

```
DOUBLE PRECISION :: value
```

Defines the value of a scalar. If both, the label and the value, are set then only the label will be written into a file.

bounds (optional): *1D-array of double*

Fortran:

```
DOUBLE PRECISION :: bounds(:)
```

Defines (two) scalar boundaries. The array size must should be equal to 2.

bounds_name (optional): *string*

Fortran:

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

Defines the name of scalar bounds as it will appear in a file. If not defined, the name will be generated automatically based on the scalar id.

prec (optional): *integer*

Fortran:

```
INTEGER :: prec
```

Defines the precision in bytes of scalar value and boundaries as it will be written into an output file. Available values are: 4 (float single precision) and 8 (float double precision). The default value is 4.

label (optional): *string*

Fortran:

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

Defines the label of a scalar. If both, the label and the value, are set then only the label will be output into a file.

scalar_ref (optional): string

Fortran:

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

Defines the reference to a scalar. All attributes will be inherited from the referenced scalar via the classical inheritance mechanism. The value assigned to the referenced scalar will be transmitted to the current scalar.

positive (optional): *enumeration {up, down}*

Fortran:

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

Defines the positive direction for fields representing height or depth.

axis_type (optional): *enumeration {X, Y, Z, T}*

Fortran:

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

Defines the type of a (scalar) axis. The values correspond to the following axis types:

- ***X:*** longitude
- **Y*:*** latitude
- **Z*:*** vertical axis
- ***T:*** time axis.

comment: *string*

Fortran:

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

Allows a user to set a comment.

1.4 Axis attribute reference

name (optional): *string*

Fortran:

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

Defines the name of a vertical axis as it will appear in an output file. If not defined, the name will be generated automatically based on the axis id. If multiple vertical axes are defined in the same file, each axis must have a unique name.

standard_name (optional): *string*

Fortran:

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

Defines the standard name of a vertical axis as it will appear in the axis' metadata in an output file.

long_name (optional): *string*

Fortran:

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

Defines the long name of a vertical axis as it will appear in the axis' metadata in an output file.

unit (optional): *string*

Fortran:

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

Defines the unit of an axis as it will appear in the axis' metadata in an output file.

dim_name (optional): *string*

Fortran:

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

Defines the name of axis dimension as it will appear in the file's metadata. The default axis dimension name is the axis name.

formula (optional): *string*

Fortran:

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

Adds the `formula` attribute to the metadata associated to the axis in the output file, for CF conformance.

formula_term (optional): *string*

Fortran:

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

Adds the `formula_term` attribute to the metadata associated to the axis in the output file, for CF conformance.

formula_bounds (optional): *string*

Fortran:

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

Adds the `formula` attribute to the metadata associated to the axis boundairies in the output file, for CF conformance. This attribute is meaningfull if `axis_bounds` is not defined.

formula_term_bounds (optional): *string*

Fortran:

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

Adds the `formula_term` attribute to the metadata associated to the axis boundairies in the output file, for CF conformance. This attribute is meaningfull if `axis_bounds` is not defined.

n_glo (mandatory): *integer*

Fortran:

```
INTEGER :: n_glo
```

Defines the global size of an axis.

begin (optional): *integer*

Fortran:

```
INTEGER :: begin
```

Defines the beginning index of the local domain. It can take value between 0 and **n_glo-1**. If not specified the default value is 0.

n (optional): *integer*

Fortran:

```
INTEGER :: n
```

Defines the local size of an axis. It can take value between 0 and **n_glo**. If not specified the default value is **n_glo**. Local axis decomposition can be declared either with attributes ***{n, begin}*** or with ***index***.

index (optional): *1D-array of double*

Fortran:

```
DOUBLE PRECISION :: index(:)
```

Defines the global indexes of a local axis held by each process. If the attribute is specified, its array size must be equal to **n**. Local axis decomposition can be declared either with attributes ***{n, begin}*** or with ***index***.

value (optional): *1D-array of double*

Fortran:

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

Defines the value of each level of a vertical axis. The array size must be equal to the value of the attribute **n**. If the label is provided then only the label will be written into a file and not the axis value and the axis boundaries.

bounds (optional): *2D-array of double*

Fortran:

```
DOUBLE PRECISION :: bounds(:,:)
```

Defines the boundaries of each level of a vertical axis. The dimensions of the array must be $2 \times n$.

bounds_name (optional): *string*

Fortran:

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

Defines the name of axis boundaries as it will appear in an ouput file. If not defined, the name will be generated automatically based on the axis id.

prec (optional): *integer*

Fortran:

```
INTEGER :: prec
```

Defines the precision in bytes of axis value and boundaries as it will be written into an output file. Available values are: 4 (float single precision) and 8 (float double precision). The default value is 4.

label (optional): *string*

Fortran:

```
CHARACTER, ALLOCATABLE :: label(:)
```

Defines the label of an axis. The size of the array must be equal to the value of the attribute **n**. If the label is provided then only the label will be written into a file and not the axis value and the axis boundaries.

data_begin (optional): *integer*

Fortran:

```
INTEGER :: data_begin
```

Defines the beginning index of local field data owned by each process. The attribute is an offset relative to the local axis, so the value can be negative. A negative value indicates that only some valid part of the data will extracted, for example in the case of a ghost cell. A positive value indicates that the local domain is greater than the data stored in memory. The 0-value means that the local domain matches the data in memory. The default value is 0. The attributes **data_begin** and **data_n** must be defined together.

data_n (optional): *integer*

Fortran:

```
INTEGER :: data_n
```

Defines the size of local field data. The attribute can take value starting from 0 (no data on a process). The default value is **n**. The attributes **data_begin** and **data_n** must be defined together.

data_index (optional): *integer*

Fortran:

```
INTEGER :: data_index
```

In case of a compressed vertical axis, the attribute defines the position of data points stored in the memory. For example, for a local axis of size **n=3** and local data size of **data_n=5**, if **data_index=(/ -1, 2, 1, 0, -1 /)** then the first and the last data points are ghosts and only the three middle values will be written in the reversed order. Only `data_begin/data_n` or `data_index` can be used together.

mask (optional): *1D-array of bool*

Fortran:

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

Defines the mask of the local axis. The masked value will be replaced by the value of the field attribute **default_value** in an output file.

n_distributed_partition (optional): *integer*

Fortran:

```
INTEGER :: n_distributed_partition
```

Defines the number of local axes in case if the axis is generated automatically by XIOS. The default value is **1**. Nota: currently this functionnality is broken, this attribute is meaningfull.

axis_ref (optional): string

Fortran:

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

Defines the reference of an axis. All attributes will be inherited from the referenced axis with the classical inheritance mechanism. The value assigned to the referenced axis will be transmitted to the current axis.

positive (optional): *enumeration {up, down}*

Fortran:

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

Defines the positive direction for fields representing height or depth. It will just be appended in axis metadata in output file, for CF compliance.

axis_type (optional): *enumeration {X, Y, Z, T}*

Fortran:

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

Defines the type of an axis. The values correspond to the following axis types:

- ***X:*** longitude
- **Y*:*** latitude
- **Z*:*** vertical axis
- ***T:*** time axis.

It will just be append in axis metadata in ouput file, for CF compliance.

comment (optional): *string*

Fortran:

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

Allows a user to set a comment.

1.5 Domain attribute reference

name (optional): *string*

Fortran:

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

Defines the name of a horizontal domain. This attribute may be used in case of multiple domains defined in the same file. In this case, the **name** attribute will be suffixed to the longitude and latitude dimensions and axis name. If the domain name is not provided, it will be generated automatically with the id of the domain.

standard_name (optional): *string*

Fortran:

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

Defines the standard name of a domain as it will appear in the domain's metadata in an output file.

long_name (optional): *string*

Fortran:

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

Defines the long name of a domain as it will appear in the domain's metadata in an output file.

type (mandatory): *enumeration {rectilinear, curvilinear, unstructured, gaussian}*

Fortran:

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

Defines the type of a grid.

dim_i_name (optional): *string*

Fortran:

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

Defines the name of the first domain dimension as it will appear in the file's metadata. The default value is '`X`'. In case of multiple domain in the file, the dimension will be preffixed by the domain name.

dim_j_name (optional): *string*

Fortran:

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

Defines the name of the second domain dimension as it will appear in file's metadata. The default value is 'Y'. In case of multiple domain in the file, the dimension will be preffixed by the domain name.

ni_glo (mandatory): *integer*

Fortran:

```
INTEGER :: ni_glo
```

Defines the size of the first dimension of the global domain.

nj_glo (mandatory): *integer*

Fortran:

```
INTEGER :: nj_glo
```

Defines the size of the second dimension of the global domain.

ibegin (optional): *integer*

Fortran:

```
INTEGER :: ibegin
```

Defines the beginning index of the first dimension of a local domain. The attribute takes value between **0** and **ni_glo-1**. If not specified the default value is **0**.

ni (optional): *integer*

Fortran:

```
INTEGER :: ni
```

Defines the size of the first dimension of a local domain. The attribute takes value between **1**and **ni_glo**. If not specified the default value is **ni_glo**.

jbegin (optional): *integer*

Fortran:

```
INTEGER :: jbegin
```

Defines the beginning index of the second dimension of a local domain. The attribute takes value between **0** and **nj_glo-1**. If not specified the default value is **0**.

nj (optional): *integer*

Fortran:

```
INTEGER :: nj
```

Defines the size of the second dimension of a local domain. he attribute takes value between **1**and **nj_glo**. If not specified the default value is **nj_glo**.

lonvalue_1d (optional): *1D-array of double*

Fortran:

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

Defines the longitude values of a local domain. For a cartesian grid, the array size should be **ni**. For a curvilinear grid, the array size should be **ni**×**nj**. In this case the first and second dimensions are collapsed into a linear array. For unstrutured and gaussian grid, the array size sould be **ni** (the second dimension **nj** is not used). Only latvalue_1d or latvalue_2d can be defined. Also the layout of latitude and longitude should be in conformance with each other: either 1D or 2D.

lonvalue_2d (optional): *2D-array of double*

Fortran:

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

Defines the longitude values of a local domain. For cartesian grid, the array size must beand curvilinear grids the array size should be **(ni,nj)**. For unstructured or gaussian grid it sould be **(ni,1)**. Only lonvalue_1d or lonvalue_2d can be defined. Also the layout of latitude and longitude should be in conformance with each other: either 1D or 2D.

latvalue_1d (optional): *1D-array of double*

Fortran:

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

Defines the latitude values of a local domain. For a cartesian and curvilinear grid, the array size should be **ni**×**nj**. In this case the first and second dimensions are collapsed into a linear array. For unstrutured and gaussian grid, the array size sould be **ni** (the second dimension **nj** is not used). Only latvalue_1d or latvalue_2d can be defined. Also the layout of latitude and longitude should be in conformance with each other: either 1D or 2D.

latvalue_2d (optional): *2D-array of double*

Fortran:

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

Defines the latitude values of a local domain. For cartesian grid and curvilinear grids the array size should be **(ni,nj)**. For unstructured or gaussian grid it sould be **(ni,1)**. Only latvalue_1d or latvalue_2d can be defined. Also the layout of latitude and longitude should be in conformance with each other: either 1D or 2D.

lon_name (optional): *string*

Fortran:

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

Define the longitude name as it will appear in an output file. If not defined, the default will be '`lon`'. In case of multiple domain defined in a file, the longitude name will be suffixed by the domain name.

lat_name (optional): *string*

Fortran:

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

efine the longitude name as it will appear in an output file. If not defined, the default will be '`lat`'. In case of multiple domain defined in a file, the longitude name will be suffixed by the domain name.

nvertex (optional): *integer*

Fortran:

```
INTEGER :: nvertex
```

Used only for unstructured domain. Defines the maximum number of vertices for a grid. The attribute is required for specifying the cell boundaries of the unstructured meshes. For other grid (cartesian, curvilinear or gaussian), the nvertex value will be set automatically to 4.

bounds_lon_1d (optional): *2D-array of double*

Fortran:

```
DOUBLE PRECISION :: bounds_lon(:,:)
```

Defines the longitude values of the domain vertexes. For unstructured domain, the attribute **nvertex** must be also defined (for other mesh it is set automatically to 4). The array dimensions shape must be of size`(nvertex, ni)`.

bounds_lon_2d (optional): *3D-array of double*

Fortran:

```
DOUBLE PRECISION :: bounds_lon(:,:,:)
```

Defines the longitude values of the domain vertexes. For unstructured domain, the attribute **nvertex** must be also defined (for other mesh it is set automatically to 4). This attribute is useful when lonvalue_2d is defined. for cartesian and curvilinear domain, the array dimensions shape must be of size `(nvertex, ni, nj)`, and for gaussian or unstructured domain, it sould be of size `(nvertex, ni, 1)`. For cartesian, curvilinear . Either `bounds_lon_1d` or `bounds_lon_2d` can be defined.

bounds_lat_1d (optional): *2D-array of double*

Fortran:

```
DOUBLE PRECISION :: bounds_lat(:,:)
```

Defines the latitude values of the domain vertexes. For unstructured domain, the attribute **nvertex** must be also defined (for other mesh it is set automatically to 4). The array dimensions shape must be of size`(nvertex, ni)`.

bounds_lat_2d (optional): *3D-array of double*

Fortran:

```
DOUBLE PRECISION :: bounds_lat(:,:)
```

Defines the latitude values of domain vertexes. For unstructured domain, the attribute **nvertex** must be also defined. This attribute is useful when lonvalue_2d is defined. for cartesian and curvilinear domain, the array dimensions shape must be of size `(nvertex, ni, nj)`, and for gaussian or unstructured domain, it sould be of size `(nvertex, ni, 1)`. For cartesian, curvilinear . Either `bounds_lon_1d` or `bounds_lon_2d` can be defined.

bounds_lon_name (optional): *string*

Fortran:

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

Defines the boundaries longitude boundaries name of domain as it will appear in an output file. Default value is '`bounds_lon`'. In case of multiple domains defined in a same file, the boundaries name will be suffixed by the domain name.

bounds_lat_name (optional): *string*

Fortran:

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

Defines the boundaries longitude boundaries name of domain as it will appear in an output file. Default value is '`bounds_lon`'. In case of multiple domains defined in a same file, the boundaries name will be suffixed by the domain name.

area (optional): *2D-array of double*

Fortran:

```
DOUBLE PRECISION :: area(:,:)
```

The area of cells. The size of the array must `(ni, nj)` for cartesian or curvilinear domain and (ni, 1) for unstructured or gaussian domain. If this attribute is defined, it will be wrote in the output file as domain metadata. This attribute could be also used in domain interpolation to take into account the area of the cell instead to take the computed value (for conservation consideration).

radius (optional): *double*

Fortran:

```
DOUBLE PRECISION :: radius
```

Define the radius of the planet (or earth). If defined, it can be used by domain interpolation together with `area` domain attribute in order to renormalize weight to compute conservative interpolation (see `domain_interpolate` filter).

prec (optional): *integer*

Fortran:

```
INTEGER :: prec
```

Defines the precision (in bytes) for which domain attributes (longitude, latitude, boudairies, ...) will be written in output file. Available values are: 4 (float single precision) and 8 (float double precision). The default value is 4 bytes.

data_dim (optional): *integer*

Fortran:

```
INTEGER :: datadim
```

Defines how a field is stored on memory for the client code. The value can be either **1** or **2**. The value of **1** indicates that the horizontal layer of the field is stored as a 1D array. The value of **2** indicates that the horizontal layer is stored as a 2D array. The default value is **1**.

data_ibegin (optional): *integer*

Fortran:

```
INTEGER :: data_ibegin
```

Defines the beginning index of field data for the first dimension. This attribute is an offset relative to the local domain, so the value can be negative. A negative value indicates that only some valid part of the data will extracted, for example in the case of a ghost cell. A positive value indicates that the local domain is greater than the data stored in memory. A 0-value means that the local domain matches the data in memory. The default value is 0. The attributes **data_ibegin** and **data_ni** must be defined together. Only data_ibegin/data_ni or data_i_index/

data_ni (optional): *integer*

Fortran:

```
INTEGER :: data_ni
```

Defines the size of field data for the first dimension. The default value is **ni**. The attributes **data_ibegin** and **data_ni** must be defined together.

data_jbegin (optional): *integer*

Fortran:

```
INTEGER :: data_jbegin
```

Defines the beginning index of field data for the second dimension. The attribute is taken into account only if **data_dim=2**. The attribute is an offset relative to 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. The 0-value means that the local domain matches the data in memory. The default value is **0**. The attributes **data_jbegin** and **data_nj** must be defined together.

data_nj (optional): *integer*

Fortran:

```
INTEGER :: data_nj
```

Defines the size of field data for the second dimension. The attribute is taken account only if **data_dim=2**. The default value is **nj**. The attributes **data_jbegin** and **data_nj** must be defined together.

data_i_index (optional): *1D-array of integer*

Fortran:

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

In case of a compressed horizontal domain, define the data indexation for the first dimension. This attribute cannot be used together with `data_ibegin/data_ni` attributes. This attribute can be used only when `data_dim=1`. The size of data_i_index determine the size of the fields dimension of the corresponding domain, that is sent from the model. Value can be negative or greater than `ni`. In this case, the corresponding point will be considered as masked and will not be extracted to enter the workflow.

data_j_index (optional): *1D-array of integer*

Fortran:

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

In case of a compressed horizontal domain, define the data indexation for the first dimension. This attribute cannot be used together with `data_jbegin/data_nj` attributes. This attribute can be used only when `data_dim=1`. The size of data_i_index determine the size of the fields dimension of the corresponding domain, that is sent from the model. Value can be negative or greater than `nj`. In this case, the corresponding point will be considered as masked and will not be extracted to enter the workflow.

mask_1d (optional): *1D-array of bool*

Fortran:

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

Defines a 1D mask of a local domain. The masked value will be replaced by the value of the field attribute **default_value** in an output file. This attribute must be of size `ni*nj`. By default no values are masked.

mask_2d (optional): *2D-array of bool*

Fortran:

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

Defines a 2D mask of a local domain. The masked values will be replaced by the value of the field attribute **default_value** in an output file. This attribute must be of size `(ni,nj)`. By default no values are masked.

domain_ref (optional): string

Fortran:

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

Defines the reference to a domain. All attributes are inherited from the referenced domain with the classic inheritance mechanism. The value assigned to the referenced domain is transmitted to to current domain.

i_index (optional): *1D-array of double*

Fortran:

```
DOUBLE PRECISION :: i_index(:)
```

Defines the global index of the first dimension of a local domain held by a process. By default the size of the array must be **ni*nj**.

j_index (optional): *1D-array of double*

Fortran:

```
DOUBLE PRECISION :: j_index(:)
```

Defines the global index of the second dimension of a local domain held by a process. By default the size of the array must be **ni*nj**.

comment (optional): *string*

Fortran:

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

Allows a user to set a comment.

1.6 Grid attribute reference

name (optional): string

Fortran:

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

Defines the name of a grid.

description (optional): string

Fortran:

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

Defines the descriptions of a grid.

mask_1d (optional): *1D-array of bool*

Fortran:

```
LOGICAL :: mask_1d(:)
```

Defines the mask of a local 1D grid. Masked values will be replaced by the value of the field attribute **default_value** in an output file. By default none of the value are masked.

mask_2d (optional): *2D-array of bool*

Fortran:

```
LOGICAL :: mask_2d(:,:)
```

Defines the mask of a local 2D grid. Masked values will be replaced by the value of the field attribute **default_value** in an output file. By default none of the value are masked.

mask_3d (optional): *3D-array of bool*

Fortran:

```
LOGICAL :: mask_3d(:,:,:)
```

Define the mask of a local 3D grid. Masked values will be replaced by the value of the field attribute **default_value** in an output file. By default none of the value are masked.

mask_4d (optional): *4D-array of bool*

Fortran:

```
LOGICAL :: mask_4d(:,:,:)
```

Define the mask of a local 4D grid. Masked values will be replaced by the value of the field attribute **default_value** in an output file. By default none of the value are masked.

mask_5d (optional): *5D-array of bool*

Fortran:

```
LOGICAL :: mask_5d(:,:,:)
```

Define the mask of a local 5D grid. Masked values will be replaced by the value of the field attribute **default_value** in an output file. By default none of the value are masked.

mask_6d (optional): *6D-array of bool*

Fortran:

```
LOGICAL :: mask_6d(:,:,:)
```

Define the mask of a local 6D grid. Masked values will be replaced by the value of the field attribute **default_value** in an output file. By default none of the value are masked.

mask_7d (optional): *7D-array of bool*

Fortran:

```
LOGICAL :: mask_7d(:,:,:)
```

Define the mask of a local 7D grid. Masked values will be replaced by the value of the field attribute **default_value** in an output file. By default none of the value are masked.

comment (optional): *string*

Fortran:

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

Allows a user to set a comment. Meaningfull for the XIOS workflow.

1.7 Field attribute reference

name (optional): *string*

Fortran:

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

Defines the name of a field as it will appear in an output file. If not present, the identifier **id** will be substituted.

standard_name (optional): *string*

Fortran:

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

Defines the **standard_name** attribute as it will appear in the metadata of an output file.

long_name (optional): *string*

Fortran:

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

Defines the long name as it will appear in the metadata of an output file.

expr (optional): *string*

Fortran:

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

Defines the expression for arithmetic or time-integration operations performed on a field. For example if **expr="sqrt(@temp2 - pow(@temp, 2))"** then the variance will be calculated on the incoming flux of the field **temp** (given that field temp2 that holds the square of temp was correctly defined).

unit (optional): *string*

Fortran:

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

Defines the unit of a field, as it will apear in the associated metadata in the output file.

operation (mandatory): enumeration *{once, instant, average, maximum, minimum, accumulate}*

Fortran:

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

Defines the temporal operation applied to a field.

freq_op (optional): *duration*

Fortran:

```
TYPE(xios_duration) :: freq_op
```

Defines the sampling frequency of a temporal operation, so that field values will be used for temporal sampling at frequency **freq_op**. It can be also useful for sub-processes called at different frequency in a model. The default value is equal to the file attribute **output_freq** for **instant** operations and **1ts** (1 time step) otherwise.

freq_offset (optional): *duration*

Fortran:

```
TYPE(xios_duration) :: freq_offset
```

Defines the offset when **freq_op** is defined. Accepted values lie between **0** and **freq_op - 1ts**. The default value is **freq_op - 1ts** for fields in the **write** mode and **0** for fields in the **read** mode.

level (optional): *integer*

Fortran:

```
INTEGER :: level
```

Defines the output level of a field. The field will be output only if the file attribute **output_level** $\geq$**level**. The default value is **0**.

prec (optional): *integer*

Fortran:

```
INTEGER :: prec
```

Defines the precision in bytes of a field in an output file. Available values are: 2 (integer), 4 (float single precision) and 8 (float double precision). The default value is 4 (float single precision).

enabled (optional): *bool*

Fortran:

```
LOGICAL :: enabled
```

Defines if a field must be output or not. The default value is **true**.

check_if_active (optional): *bool*

Fortran:

```
LOGICAL :: check_if_active
```

Sets a check if a field will be used at a given time step. Activating the check may improve performance for fields which are not used frequently, while it can detoriate performance for fields used at each time step. The default value is false.

read_access (optional): *bool*

Fortran:

```
LOGICAL :: read_access
```

Defines whether a field can be read from the model or not. The default value is **false**. Note that for fields belonging to a file in ***read* mode**, this attribute is always **true**.

field_ref (optional): *string*

Fortran:

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

Defines the field reference. All attributes will be inherited from the referenced field via the classical inheritance mechanism. The values assigned to the referenced field will be transmitted to the current field to perform temporal operation.

grid_ref (optional): *string*

Fortran:

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

Defines the field grid. Note that only either **grid_ref** or a combination of **domain_ref**, **scalar_ref** or **axis_ref** can be specified.

domain_ref (optional): *string*

Fortran:

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

Defines the field domain. If the attribute is defined, the attribute **grid_ref** must not be specified.

axis_ref (optional): *string*

Fortran:

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

Defines an axis for the current field. If the attribute is defined, the attribute **grid_ref** must not be specified.

scalar_ref (optional): *string*

Fortran:

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

Defines a scalar domain for the current field. If the attribute is defined, the attribute **grid_ref** must not be specified.

grid_path (optional): *string*

Fortran:

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

Defines the way operations pass from a grid to other grids, combined with field_ref attribute. Each grid name in `grid_path` must be separated by a comma. All tranformation realated to the grid listed in grid_path will be applyed successively, including the grid destination associated to the destination field.

default_value (optional): *double*

Fortran:

```
DOUBLE PRECISION :: default_value
```

Defines the value which will be used instead of missing field data. For an entry field (coming from model, or from an input file), if `detect_missing_value` and `default_value` are defined the values of the field equal to `default_value` will be replaced by NaN. For output field (read from the model or sent to server), the NaN values will be replaced by `default_value`, if defined. For masked value, in output, if `default_value` is defined, the corresponding value will be replaced by `default_value`, otherwise, the value is undefined and will depend of the initial value of the allocated memory block.

valid_min (optional): *double*

Fortran:

```
DOUBLE PRECISION :: valid_min
```

Define the minimum validity range of the field as it will be written as a field metadata in the output file, accordingly to CF compliance.

valid_max (optional): *double*

Fortran:

```
DOUBLE PRECISION :: valid_max
```

Define the maximum validity range of the field as it will be written as a field metadata in the output file, accordingly to CF compliance.

detect_missing_value (optional): *bool*

Fortran:

```
LOGICAL: detect_missing_value
```

For entry field, if `detect_missing_value` is set to true and `default_value` is defined, the corresponding value will be replaced by NaN (see `default_value` attribute). This attribute is also used during temporal operation such as averaging, minimum, maximum, accumulate..., to detect NaN value and remove it from the computation (otherwise the result will be NaN). It is used also in some spatial transformation, but will be replaced later by a specific detect_missing value attribute related to the transformation.

add_offset (optional): *double*

Fortran:

```
DOUBLE PRECISION: add_offset
```

Sets the **add_offset** metadata CF attribute in an output file. In output, the **add_offset** value will be subtracted from the field values. It can be used together with `scale_factor` and `prec` attributes in order to reduce the data range and the volume of written data. Fo reading, this attribute is also taking into account to "decompress" data.

scale_factor: *double*

Fortran:

```
DOUBLE PRECISION: scale_factor
```

Sets the **scale_factor** metadata CF attribute in an output file. In output, the field values will be divided by the **scale_factor** value. It can be used together with `add_offset` and `prec` attributes in order to reduce the data range and the volume of written data. Fo reading, this attribute is also taking into account to "decompress" data.

compression_level (optional): *integer*

Fortran:

```
INTEGER :: compression_level
```

Defines whether a field should be compressed using NetCDF-4 built-in compression. The compression level must range from 0 to 9. A higher compression level means a better compression at the cost of using more processing power. The default value is inherited from the file attribute **compression_level**.

indexed_output (optional): *bool*

Fortran:

```
LOGICAL :: indexed_output
```

Defines whether field data must be output as an indexed grid instead of a full grid whenever possible. The default value is ***false***.

ts_enabled (optional): *bool*

Fortran:

```
LOGICAL :: ts_enabled
```

Defines whether a field can be output as a timeseries. The default value is ***false***. A field will be tagged as timeseries if `ts_enabled` is set to true, and the associated file attribute timeserie is set to `only`, `both` or `exclusive`. A field tagged as timeseries will be output alone in a specific file suffixed by the field name.

ts_split_freq (optional): *duration*

Fortran:

```
TYPE(xios_duration) :: ts_split_freq
```

Defines the splitting frequency that should be used for a timeseries if it has been requested. By default the attribute value is inherited from the file attribute **split_freq**.

cell_methods (optional): *string*

Fortran:

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

Defines the cell methods field attribute. it will be written as associated field metadata, accordingly to CF compliance.

cell_methods_mode (optional): enumeration *{overwrite, prefix, suffix, none}*

Fortran:

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

Defines the cell methods mode of a field.

XIOS defined itself the cell_method string, taking into account the operations applied on the field. But in some case it can modified, with the `cell_methods` attribute. `overwrite` will replace the computed default string by the `cell_methods` attribute, `prefix` will prefix the default string, and `suffix` will suffix the default string.

comment (optional): *string*

Fortran:

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

Allows a user to set a comment. Meaningfull on XIOS workflow.

1.8 Variable attribute reference

name (optional): *string*

Fortran:

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

Defines the name of a variable as it will appear in an output file. If not present, the variable **id** will be used.

type (mandatory): enumeration {bool, int, int32, int16, int64, float, double, string}

Fortran:

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

Defines the type of a variable. Note that the ***int*** type is a synonym for ***int32***.

ts_target (optional): enumeration {file, field, both, none}

Fortran:

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

This attribute specify if a variable present in a file (global attribute) must be reported in each timeseries generated file. If `ts_target=none`, the variable is not included, if `ts_target=file`, the variable is included in each generated file as global attribute, if `ts_target=field`, the variable is included as field attribute associated to the generated files (field metadata), if `ts_target=both`, the varible is included twice as file global attribute and field attribute.

1.9 File attribute reference

name (optional): *string*

Fortran:

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

Defines the name of a file. If not specified a name is automaticaly defined from the file id. For netcdf output, a "`.nc`" suffix will be added to the file name.

description (optional): *string*

Fortran:

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

Defines the description of a file.

name_suffix (optional): *string*

Fortran:

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

Defines a suffix added to the file name.

min_digits (optional): *integer*

Fortran:

```
INTEGER :: min_digits
```

For the **multiple_file** mode defines the minimum number of digits of the suffix describing the server rank which will be appended to the file name. The default value is **0** (no server rank suffix is added).

output_freq (mandatory): *duration*

Fortran:

```
TYPE(xios_duration) :: output_freq
```

Defines the output frequency for the current file.

output_level (optional): *integer*

Fortran:

```
INTEGER :: output_level
```

Defines the output level for all fields of the current file. The field is output only if the field attribute **level** is less or equal to the file attribute **output_level**.

sync_freq (optional): *duration*

Fortran:

```
TYPE(xios_duration) :: sync_freq
```

Defines the frequency for flushing the current file onto a disk. It may result in poor performances but data will be written even if a file is not yet closed.

split_freq (optional): *duration*

Fortran:

```
TYPE(xios_duration) :: split_freq
```

Defines the frequency for splitting the current file. The start and end dates will be added to the file name (see **split_freq_format** attribute). By default no splitting is done.

split_freq_format (optional): *string*

Fortran:

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

Defines the format of the split date suffixed to a file. It can contain any character, `%y` will be replaced by the year (4 characters), `%mo` by the month (2 char), `%d` by the day (2 char), `%h` by the hour (2 char), `%mi` by the minute (2 char), `%s` by the second (2 char), `%S` by the number of seconds since the time origin and `%D` by the number of full days since the time origin. 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 not appear in the suffix, only year, month and day.

split_start_offset (optional): *duration*

Fortran:

```
TYPE(xios_duration) :: split_start_offset
```

Defines the offset of the start splitting date when `split_freq` attribute is enabled. This attribute is used only to shift the start date of the splitting format. It has no impact on the real splitting date.

split_end_offset(optional): *duration*

Fortran:

```
TYPE(xios_duration) :: split_end_offset
```

Defines the offset of the end splitting date when `split_freq` attribute is enabled. This attribute is used only to shift the end date of the splitting format. It has no impact on the real splitting date.

split_last_date (optional): *string*

Fortran:

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

Define the end splitting date when `split_freq` attribute is enabled. If `split_last_date` is lesser than the compute end splitting date, then the end splitting date will be replaced by `split_last_date` in the file name. This attribute can be usefull when a simulation is over before the computed splitting date, to be consistent with the file internal calendar. The format of `split_last_date` string attribute must be consistent with a date.

enabled (optional): *bool*

Fortran:

```
LOGICAL :: enabled
```

Defines if a file must be written/read or not. The default value is **true**.

mode (optional): *enumeration {read, write}*

Fortran:

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

Defines whether a file will be read or written. The default value is **write**.

type (optionnal): *enumeration {one_file, multiple_file}*

Fortran:

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

Defines the type of a file: ***multiple_file***: one file by server using sequential netcdf writing, ***one_file***: one single global file is wrote using netcdf4 parallel access. The default value is `multiple_file`.

format (optional): *enumeration {netcdf4, netcdf4_classic}*

Fortran:

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

Define the format of a file: ***netcdf4***: the HDF5 format will be used, ***netcdf4_classic***: the classic NetCDF format will be used. The default value is ***netcdf4***. Note that the ***netcdf4_classic*** format can be used with the attribute **type** set to ***one_file*** only if the NetCDF4 library was compiled with Parallel NetCDF support (–enable-pnetcdf).

par_access (optional): *enumeration {collective, independent}*

Fortran:

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

For parallel writing, defines which type of MPI calls will be used. The default value is ***collective***. Nota : this attribute is now deprecated, and has no effect on I/O parallelism.

read_metadata_par (optional): *bool*

Fortran:

```
LOGICAL :: read_metadata_par
```

For files in the read mode, defines if parallel or serial I/O will be used by model processes for reading file metadata. The default value is false implying serial I/O for reading metadata on client side.

convention (optional): *enumeration {CF, UGRID}*

Fortran:

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

Defines the file conventions. The netcdf file can be output following the CF convention (1.7) or the UGRID convention, usefull for unstructured mesh. By default the CF conventions are followed. The global file attribute `Conventions` will be added in the output file following the specified value.

convention_str (optional): *string*

Fortran:

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

Defines the **Conventions** attribute to be added to file global attributes. This attribute will overwrite the `Conventions` global file attribute generated by the `convention` XIOS attribute.

append (optional): *bool*

Fortran:

```
LOGICAL :: append
```

If this attribute is set to `true` and if the file is present, the file is open, and data will be append at the specified time record. If the time record already exists, data will be overwritten, otherwise a new time record is created and data is appended. If append is set to `false`, a new file will be created, even if a old file is present, overwritting it. The default value is `append = false`.

compression_level (optional): *integer*

Fortran:

```
INTEGER :: compression_level
```

Defines whether the fields should be compressed using NetCDF-4 built-in compression by default. The compression level must range from 0 to 9. A higher compression level means a better compression at the cost of using more processing power. The default value is ***0*** (no compression).

time_counter (optional): *enumeration {centered, instant, record, exclusive, centered_exclusive, instant_exclusive, none}*

Fortran:

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

Defines how the "time_counter" variable will be output:

- ***centered***: use centered times (default option for all field operations except for instant)
- ***instant***: use instant times (default option for field operation **instant**)
- ***record***: use record indexes
- ***centered_exclusive:*** do not include centered times into an output file
- **instant**_***exclusive:*** do not include instant times into an output file
- ***exclusive:*** include neither instant times nor centered times into an output file
- ***none***: do not output the variable.

The default value is `centered` if there are only centered time axis. It will be `instant` if there are only instant time axis. If both instant and centered time axis are present, the default value will be `centered`.

time_counter_name (optional): *string*

Fortran:

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

Define the name of a time counter. The default value is `time_counter`.

timeseries (optional): *enumeration {none, only, both, exclusive}*

Fortran:

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

Defines whether the timeseries must be output:

- ***none***: no timeseries is outputted, only the regular file
- ***only***: only the timeseries is outputted, the regular file is not created
- ***both***: both the timeseries and the regular file are outputted.
- ***exclusive***: the timeseries is outputted and a regular file is created with only the fields which were not marked for output as a timeseries (if any).

When time series is enabled, each tagged field (see `ts_enabled` field attribute) is output into a separate file. The default value is ***none***.

ts_prefix (optional): *string*

Fortran:

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

Defines the prefix to use for the name of the timeseries files. By default the file name will be used.

time_units (optional): *enumeration {seconds, days}*

Fortran:

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

Define the time unit of the time axis present in the file. Two choices are possible : `seconds` or `days`. The default value is `seconds`.

record_offset (optional): *integer*

Fortran:

```
INTEGER :: record_offset
```

Defines an offset for the time records associated to the file. For a file in `read` mode, the first record sent will be shift of`record_offset` value instead of 0. For file in write mode, the first time record to be output will begin from `record_offset` value. The default value is 0.

cyclic (optional): *bool*

Fortran:

```
LOGICAL :: cyclic
```

If the option is activated for fields to be read, then upon reaching the last time record, reading will continue "cycle" at the first time record. The default value is false.

time_stamp_name (optional): *string*

Fortran:

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

Defines the timestamp name of the date and time when the program was executed which will be written into an output file. The default value is "timeStamp".

time_stamp_format (optional): *string*

Fortran:

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

Defines the timestamp format of the date and time when the program was executed to be written into an output file. It can contain any character. `%Y` will be replaced by the 4-digit year (4 digits), while `%y` will be replaced by the 2-digit year. `%m` will be by the 2-digit month, while %b will be replaced by the 3-character month. `%d` will be replaced by the day (2 char), `%H` by the hour (2 char), `%M` by the minute (2 char), `%S` by the number of seconds, `%D` by the date in the MM/DD/YY format.

uuid_name (optional): *string*

Fortran:

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

Defines the name of the global attribute designing the UUID of the file. The default global attribute is "`uuid`".

uuid_format (optional): *string*

Fortran:

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

Defines the format of file's UUID. In the `uuid_format` string, the sub-string `%uuid%` will be replaced by string computed UUID.

comment (optional): *string*

Fortran:

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

Allows a user to set a comment.

1.10 Transformation attribute reference

1.10.1 reduce_scalar_to_scalar

operation (mandatory): *enumeration {min, max, sum, average}*

Fortran:

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

Defines the type of reduction operation performed on the scalar. Like a scalar is redondant accross process (except if masked), the reduction will be similar to a MPI reduction accros procces.

1.10.2 extract_axis_to_scalar

position: *integer*

Fortran:

```
INTEGER :: position
```

Global index of a point on an axis to be extracted into a scalar.

1.10.3 interpolate_axis

type (optional): *string*

Fortran:

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

Defines the interpolation type on an axis. For now only polynomial interpolation is available, so this attribute is currently meaningless.

order (optional): *integer*

Fortran:

```
INTEGER :: order
```

Defines the order of interpolation. The default value is 2.

coordinate (optional): *string*

Fortran:

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

Defines the coordinate associated with an axis on which interpolation will be performed.

1.10.4 reduce_axis_to_axis

operation (mandatory): *enumeration {min, max, sum, average}*

Fortran:

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

Defines a reduction operation performed on an axis across model processes.

1.10.5 reduce_axis_to_scalar

Reduces data defined on an axis into a scalar value.

operation (mandatory): *enumeration {min, max, sum, average}*

Fortran:

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

1.10.6 zoom_axis

begin (optional): *integer*

Fortran:

```
INTEGER :: begin
```

Defines the beginning index of a zoomed region on a global axis. The attribute value should be an integer between **0** and **ni_glo-1** of the associated axis. If not specified the default value is **0**.

n (optional): *integer*

Fortran:

```
INTEGER :: n
```

Defines the size of a zoomed region on a global axis. The attribute value should be an integer between **1**and **nj_glo** of the associated axis. If not specified the default value is **nj_glo** of the associated axis.

1.10.7 compute_connectivity_domain

n_neighbor: *1D-array of integer*

Fortran:

```
INTEGER :: n_neighbor(:)
```

local_neighbor: *2D-array of integer*

Fortran:

```
INTEGER :: local_neighbor(:,:)
```

n_neighbor_max: *integer*

Fortran:

```
INTEGER :: n_neighbor_max
```

1.10.8 extract_domain_to_axis

position (optional): *integer*

Fortran:

```
INTEGER :: position
```

Defines the index on a domain starting which an axis will be extracted along the direction specified with the **direction** attribute.

direction (mandatory): *enumeration {iDir, jDir}*

Fortran:

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

Defines the domain dimension along which an axis will be extracted. `iDir` means along the direction i, `jDir` along the direction j.

1.10.9 interpolate_domain

order (optional): *integer*

Fortran:

```
INTEGER :: order
```

Defines the order of interpolation. This attribute is only for internal interpolation module. Currently, only first order and second order are available for conservative interpolations. The default value is 2.

renormalize (optional): *bool*

Fortran:

```
LOGICAL :: renormalize
```

This flag is usefull only when targeted cells intersect masked source cells. In case a field corresponding to a flux value, the result of the interpolation remain correct. Otherwise for intensive field, the computed value need to be renormalize prorate of the non masked intersected area to remain correct. This flag activate these functionnality. Default value is `false`.

quantitity (optional): *bool*

Fortran:

```
LOGICAL :: quantity
```

If this flag is set, the interpolation will preserve extensive property of the field. Default value is `false`.

use_area (optional): *bool*

Fortran:

```
LOGICAL :: use_area
```

If this flag is set, given area for source and target domain (if any) will be used to renormalize compute weight by the ratio given area / computed area. Default value is `false`.

detect_missing_value (optional): *bool*

Fortran:

```
LOGICAL :: detect_missing_value
```

If flag is set to true, input data of the field to be interpolated are analyzed to detect missing values. Detected cells are considered as masked and will be taking into account for interpolation in a similar way than the domain mask. Default value is `false`.

mode (optionnal): *enumeration {compute, read, read_or_compute}*

Fortran:

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

Define if interpolation must be computed (compute), read from file (read), or if find file weight, read otherwise compute (read_or_compute). Default value is `compute`.

write_weight (optional): *bool*

Fortran:

```
LOGICAL :: write_weight
```

If this flag is set, the computed weights will be written in file. If weights are already read from a file, nothing will happen. Default value is `false`.

weight_filename (optional): *string*

Fortran:

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

Define the file name where the weights will be written or read. If not specified when trying to read or write, a name will be automatically generated.

read_write_convention (optionnal): *enumeration {c, fortran}*

Fortran:

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

Define the convention for pair of global index source and destination for wich a weight is defined. For C convention, index will begin to 0, for fortran, index will begin to 1. Usefull only for read ar write weights from/to a file. Default value is corresponding to C convention.

1.10.10 reduce_domain_to_axis

direction: *enumeration {iDir, jDir}*

Fortran:

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

Defines the domain dimension along which a reduction of the domain into an axis will be performed. `iDir` means along the direction i, `jDir` along the direction j.

operation (mandatory): *enumeration {min, max, sum, average}*

Fortran:

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

Defines the reduction operation performed on the domain.

local (optionnal): *bool*

Fortran:

```
LOGICAL :: local
```

Defines whether the reduction should be performed locally on data owned by each process or on the global domain.

1.10.11 reduce_domain_to_scalar

Reduces data defined on a domain into a scalar value.

operation (mandatory): *enumeration {min, max, sum, average}*

Fortran:

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

Defines the reduction operation performed on the domain.

local (optionnal): *bool*

Fortran:

```
LOGICAL :: local
```

Defines whether the reduction should be performed locally on data owned by each process or on the global domain. The default value is `false`.

1.10.12 reorder_domain

invert_lat (optional): *bool*

Fortran:

```
LOGICAL :: invert_lat
```

Defines whether the latitude should be inverted. The default value is false.

shift_lon_fraction (optional): *double*

Fortran:

```
DOUBLE PRECISION :: shift_lon_fraction
```

Defines the longitude offset. The value of the parameter represents a fraction of **ni_glo**. Default value is 0.

min_lon (optional): *double*

Fortran:

```
DOUBLE PRECISION :: min_lon
```

If both, **min_lon** and **max_lon**, are defined, a domain will be reordered with latitude values starting from **min_lon** and ending at **max_lon**.

max_lon (optional): *double*

Fortran:

```
DOUBLE PRECISION :: max_lon
```

If both, **min_lon** and **max_lon**, are defined, a domain will be reordered with latitude values starting from **min_lon** and ending at **max_lon**.

1.10.13 expand_domain

order: *integer*

Fortran:

```
INTEGER :: order
```

Define the size of the halo expansion.

type (optional): *enumeration {node, edge}*

Fortran:

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

Defines whether the node or edge connectivity should be calculated for the expanded domain.

i_periodic (optional): *bool*

Fortran:

```
LOGICAL :: i_periodic
```

If the attribute value is true, values of fields defined on the expanded domain will be duplicated from those of the original domain periodically along the first dimension. The default value is false (masked values on the expanded domain).

j_periodic (optional): *bool*

Fortran:

```
LOGICAL :: j_periodic
```

If the attribute value is true, values of fields defined on the expanded domain will be duplicated from those of the original domain periodically along the second dimension. The default value is false (masked values on the expanded domain).

1.10.14 zoom_domain

ibegin (optional): *integer*

Fortran:

```
INTEGER :: ibegin
```

Defines the beginning index of the zoomed region on the first dimension of the global domain. This must be an integer between **0** and **ni_glo-1** of the associated dimension of domain. If not specified the default value is **0**. Note that if one of the zoom attributes (ibegin, ni, jbegin or nj) is defined then all the rest should be specified by a user as well.

ni (optional): *integer*

Fortran:

```
INTEGER :: ni
```

Define the size of zoomed region on the first dimension of the global domain. This must be an integer between **1**and **ni_glo** of the associated dimension of domain. If not specified the default value is **ni_glo** of the dimension of domain. Note that if one of the zoom attributes (ibegin, ni, jbegin or nj) is defined then all the rest should be specified by a user as well.

jbegin (optional): *integer*

Fortran:

```
INTEGER :: jbegin
```

Define the beginning index of the zoomed region on the second dimension of the global domain. This must be an integer between **0** and **nj_glo-1** of the associated dimension of domain. If not specified the default value is **0**. Note that if one of the zoom attributes (ibegin, ni, jbegin or nj) is defined then all the rest should be specified by a user as well.

nj (optional): *integer*

Fortran:

```
INTEGER :: nj
```

Define the size of zoomed region on the second dimension of the global domain. The attribute value should be an integer between **1**and **nj_glo** of the associated dimension of domain. If not specified the default value is **nj_glo** of the dimension of domain. Note that if one of the zoom attributes (ibegin, ni, jbegin or nj) is defined then all the rest should be specified by a user as well.

1.10.15 generate_rectilinear_domain

lon_start (optional): *double*

Fortran:

```
DOUBLE PRECISION :: lon_start
```

Along with **lon_end**, the attribute defines the longitude range of a generated domain.

lon_end (optional): *double*

Fortran:

```
DOUBLE PRECISION :: lon_end
```

Along with **lon_start**, the attribute defines the longitude range of a generated domain.

lat_start (optional): *double*

Fortran:

```
DOUBLE PRECISION :: lat_start
```

Along with **lat_end**, the attribute defines the latitude range of a generated domain.

lat_end (optional): *double*

Fortran:

```
DOUBLE PRECISION :: lat_end
```

Along with **lat_start**, the attribute defines the latitude range of a generated domain.

bounds_lon_start: *double*

Fortran:

```
DOUBLE PRECISION :: bounds_lon_start
```

Attributes **bounds_lon_start** and **bounds_lon_start** set the longitude range of a generated domain. If both sets, **(lon_start, lon_end)** and **(bounds_lon_start, bounds_lon_end)**, are specified then the bound attributes will be ignored.

bounds_lon_end: *double*

Fortran:

```
DOUBLE PRECISION :: bounds_lon_end
```

Attributes **bounds_lon_start** and **bounds_lon_start** set the longitude range of a generated domain. If both sets, **(lon_start, lon_end)** and **(bounds_lon_start, bounds_lon_end)**, are specified then the bound attributes will be ignored.

bounds_lat_start: *double*

Fortran:

```
DOUBLE PRECISION :: bounds_lat_start
```

Attributes **bounds_lat_start** and **bounds_lat_start** set the latitude range of a generated domain. If both sets, **(lat_start, lat_end)** and **(bounds_lat_start, bounds_lat_end)**, are specified then the bound attributes will be ignored.

bounds_lat_end: *double*

Fortran:

```
DOUBLE PRECISION :: bounds_lat_end
```

Attributes **bounds_lat_start** and **bounds_lat_start** set the latitude range of a generated domain. If both sets, **(lat_start, lat_end)** and **(bounds_lat_start, bounds_lat_end)**, are specified then the bound attributes will be ignored.

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 split 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 successful 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 enables 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 elements currently available 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(xios_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(xios_element), INTENT(OUT):: handle
```

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

Arguments:

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

Description:

This subroutine returns the handle of the specified element identified by its string. The element must be existing otherwise an error is raised.

Query for a valid element

Synopsis:

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

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

Arguments:

- `id`: string identifier.

Description:

This function returns .TRUE. if the element defined by the string identifier "id" exists in the data base, otherwise it returns .FALSE. .

Adding child

Synopsis:

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

where element is one of the existing elements or element groups.

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 adds a child to an existing parent element. The identifier of the child, if existing, can be specified optionally. All group elements can contain child of the same type, provided generic inheritance. Some elements can contain children of another type for a specific behavior. File element may contain field_group, field, variable and variable_group child elements. Field elements may contain variable_group of variable child element.

Query if the value of an element attribute is defined (by handle)

Synopsis:

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

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

where element is one of the existing elements or element groups. 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 can 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 documentation.

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(OUT) :: attr_1
LOGICAL, OPTIONAL  , INTENT(OUT) :: attr_2
....
```

where element is one of the existing elements or element groups. 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 can 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 documentation.

Setting element attributes value by handle

Synopsis:

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

```
TYPE(xios_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 elements or element groups. 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 can be used to set one or more attributes of an element defined by its handle. The list of available attributes and their types are described in corresponding chapters of the documentation.

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 elements or element groups. The attributes attribute_x and attribute_type_x are described in corresponding chapters.

Arguments:

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

Description:

This subroutine can be used to set one or more attributes of an element defined by its string id. The list of available attributes and their type are described in corresponding chapters of the documentation.

Getting element attributes value (by handle)

Synopsis:

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

```
TYPE(xios_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 elements or element groups. 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 can 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 documentation.

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 elements or element groups. 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 can 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 documentation.

Context management interface

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 initializes 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 called to close a context before the `xios_finalize` call. It waits until that all pending requests sent to the servers will be processed and all opened files will be closed.

Setting current active context

Synopsis:

```
SUBROUTINE xios_set_current_context(context_handle)
TYPE(xios_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 following XIOS calls will refer to this active context. If only one context is defined, it will be set automatically as the active context.

Closing definition

Synopsis:

```
SUBROUTINE xios_close_context_definition()
```

Arguments:

None

Description:

This subroutine must be called when all definitions of a context are 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 undefined effect.

Calendar management interface

Creating the calendar

Synopsis:

```
SUBROUTINE xios_define_calendar(type, timestep, start_date, time_origin, &
                                day_length, month_lengths, year_length, &
                                leap_year_month, leap_year_drift, &
                                leap_year_drift_offset)
CHARACTER(len = *),              INTENT(IN) :: type
TYPE(xios_duration),   OPTIONAL, INTENT(IN) :: timestep
TYPE(xios_date),       OPTIONAL, INTENT(IN) :: start_date
TYPE(xios_date),       OPTIONAL, INTENT(IN) :: time_origin
INTEGER,               OPTIONAL, INTENT(IN) :: day_length
INTEGER,               OPTIONAL, INTENT(IN) :: month_lengths(:)
INTEGER,               OPTIONAL, INTENT(IN) :: year_length
DOUBLE PRECISION,      OPTIONAL, INTENT(IN) :: leap_year_drift
DOUBLE PRECISION,      OPTIONAL, INTENT(IN) :: leap_year_drift_offset
INTEGER,               OPTIONAL, INTENT(IN) :: leap_year_month
```

Arguments:

- `type`: the calendar type, one of `"Gregorian"`, `"Julian"`, `"D360"`, `"AllLeap"`, `"NoLeap"`, `"user_defined"`
- `timestep`: the time step of the simulation (optional, can be set later)
- `start_date`: the start date of the simulation (optional, `xios_date(0000, 01, 01, 00, 00, 00)` is used by default)
- `time_origin`: the origin of the time axis (optional, `xios_date(0000, 01, 01, 00, 00, 00)` is used by default)
- `day_length`: the length of a day in seconds (mandatory when creating an user defined calendar, must not be set otherwise)
- `month_lengths`: the length of each month of the year in days (either `month_lengths` or `year_length` must be set when creating an user defined calendar, must not be set otherwise)
- `year_length`: the length of a year in seconds (either `month_lengths` or `year_length` must be set when creating an user defined calendar, must not be set otherwise)
- `leap_year_drift`: the yearly drift between the user defined calendar and the astronomical calendar, expressed as a fraction of day (can optionally be set when creating an user defined calendar in which case `leap_year_month` must be set too)
- `leap_year_drift_offset`: the initial drift between the user defined calendar and the astronomical calendar at the time origin, expressed as a fraction of day (can optionally be set if `leap_year_drift` and `leap_year_month` are set)
- `leap_year_month`: the month to which an extra day must be added in case of leap year (can optionally be set when creating an user defined calendar in which case `leap_year_drift` must be set too)

For a more detailed description of those arguments, see the description of the corresponding attributes in section 1.2 “Calendar attribute reference”.

Description:

This subroutine creates the calendar for the current context. Note that the calendar is created once and for all, either from the XML configuration file or the Fortran interface. If it was not created from the configuration file, then this subroutine must be called once and only once before the context definition is closed. The calendar features can be used immediately after the calendar was created.

If an user defined calendar is created, the following arguments must also be provided:`day_length` and either `month_lengths` or `year_length`. Optionally it is possible to configure the user defined calendar to have leap years. In this case, `leap_year_drift` and `leap_year_month` must also be provided and `leap_year_drift_offset` might be used.

Accessing the calendar type of the current calendar

Synopsis:

```
SUBROUTINE xios_get_calendar_type(calendar_type)
CHARACTER(len=*), INTENT(OUT) :: calendar_type
```

Arguments:

- `calendar_type`: on output, the type of the calendar attached to the current context

Description:

This subroutine gets the calendar type associated to the current context. It will raise an error if used before the calendar was created.

Accessing and defining the time step of the current calendar

Synopsis:

```
SUBROUTINE xios_get_timestep(timestep)
TYPE(xios_duration), INTENT(OUT) :: timestep
```

and

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

Arguments:

- `timestep`: a duration corresponding to the time step of the simulation

Description:

Those subroutines respectively gets and sets the time step associated to the calendar of the current context. Note that the time step must always be set before the context definition is closed and that an error will be raised if the getter subroutine is used before the time step is defined.

Accessing and defining the start date of the current calendar

Synopsis:

```
SUBROUTINE xios_get_start_date(start_date)
TYPE(xios_date), INTENT(OUT) :: start_date
```

and

```
SUBROUTINE xios_set_start_date(start_date)
TYPE(xios_date), INTENT(IN) :: start_date
```

Arguments:

- `start_date`: a date corresponding to the beginning of the simulation

Description:

Those subroutines respectively gets and sets the start date associated to the calendar of the current context. They must not be used before the calendar was created.

Accessing and defining the time origin of the current calendar

Synopsis:

```
SUBROUTINE xios_get_time_origin(time_origin)
TYPE(xios_date), INTENT(OUT) :: time_origin
```

and

```
SUBROUTINE xios_set_time_date(time_origin)
TYPE(xios_date), INTENT(IN) :: time_origin
```

Arguments:

- `start_date`: a date corresponding to the origin of the time axis

Description:

Those subroutines respectively gets and sets the origin of time associated to the calendar of the current context. They must not be used before the calendar was created.

Updating the current date of the current calendar

Synopsis:

```
SUBROUTINE xios_update_calendar(step)
INTEGER, INTENT(IN) :: step
```

Arguments:

- `step`: the current iteration number

Description:

This subroutine sets the current date associated to the calendar of the current context based on the current iteration number: $current\_date = start\_date + step \times timestep$. It must not be used before the calendar was created.

Accessing the current date of the current calendar

Synopsis:

```
SUBROUTINE xios_get_current_date(current_date)
TYPE(xios_date), INTENT(OUT) :: current_date
```

Arguments:

- `current_date`: on output, the current date

Description:

This subroutine gets the current date associated to the calendar of the current context. It must not be used before the calendar was created.

Accessing the year length of the current calendar

Synopsis:

```
INTEGER FUNCTION xios_get_year_length_in_seconds(year)
INTEGER, INTENT(IN) :: year
```

Arguments:

- `year`: the year whose length is requested

Description:

This function returns the duration in seconds of the specified year, taking leap years into account based on the calendar of the current context. It must not be used before the calendar was created.

Accessing the day length of the current calendar

Synopsis:

```
INTEGER FUNCTION xios_get_day_length_in_seconds()
```

Arguments: None

Description:

This function returns the duration in seconds of a day, based on the calendar of the current context. It must not be used before the calendar was created.

Duration handling interface

Duration constants

Some duration constants are available to ease duration handling:

- `xios_year`

- `xios_month`
- `xios_day`
- `xios_hour`
- `xios_minute`
- `xios_second`
- `xios_timestep`

Arithmetic operations on durations

The following arithmetic operations on durations are available:

- Addition: `xios_duration = xios_duration + xios_duration`
- Subtraction: `xios_duration = xios_duration - xios_duration`
- Multiplication by a scalar value: `xios_duration = scalar * xios_duration` or `xios_duration = xios_duration * scalar`
- Negation: `xios_duration = -xios_duration`

Comparison operations on durations

The following comparison operations on durations are available:

- Equality: `LOGICAL = xios_duration == xios_duration`
- Inequality: `LOGICAL = xios_duration /= xios_duration`

Interface relative to date handling

Arithmetic operations on dates

The following arithmetic operations on dates are available:

- Addition of a duration: `xios_date = xios_date + xios_duration`
- Subtraction of a duration: `xios_date = xios_date - xios_duration`
- Subtraction of two dates: `xios_duration = xios_date - xios_date`

Comparison operations on dates

The following comparison operations on dates are available:

- Equality: `LOGICAL = xios_date == xios_date`
- Inequality: `LOGICAL = xios_date /= xios_date`
- Less than: `LOGICAL = xios_date < xios_date`
- Less or equal: `LOGICAL = xios_date <= xios_date`
- Greater than: `LOGICAL = xios_date > xios_date`
- Greater or equal: `LOGICAL = xios_date >= xios_date`

Converting a date to a number of seconds since the time origin

Synopsis:

```
FUNCTION INTEGER(kind = 8) xios_date_convert_to_seconds(date)
TYPE(xios_date), INTENT(IN) :: date
```

Arguments:

- `date`: the date to convert

Description:

This function returns the number of seconds since the time origin for the specified date, based on the calendar of the current context. It must not be used before the calendar was created.

Converting a date to a number of seconds since the beginning of the year

Synopsis:

```
FUNCTION INTEGER xios(date_get_second_of_year)(date)
TYPE(xios_date), INTENT(IN) :: date
```

Arguments:

- `date`: the date to convert

Description:

This function returns the number of seconds since the beginning of the year for the specified date, based on the calendar of the current context. It must not be used before the calendar was created.

Converting a date to a number of days since the beginning of the year

Synopsis:

```
FUNCTION DOUBLE_PRECISION xios_date_get_day_of_year(date)
TYPE(xios_date), INTENT(IN) :: date
```

Arguments:

- `date`: the date to convert

Description:

This function returns the number of days since the beginning of the year for the specified date, based on the calendar of the current context. It must not be used before the calendar was created.

Converting a date to a fraction of the current year

Synopsis:

```
FUNCTION DOUBLE_PRECISION xios_date_get_fraction_of_year(date)
TYPE(xios_date), INTENT(IN) :: date
```

Arguments:

- `date`: the date to convert

Description:

This function returns the fraction of year corresponding to the specified date, based on the calendar of the current context. It must not be used before the calendar was created.

Converting a date to a number of seconds since the beginning of the day

Synopsis:

```
FUNCTION INTEGER xios(date_get_second_of_day)(date)
TYPE(xios_date), INTENT(IN) :: date
```

Arguments:

- `date`: the date to convert

Description:

This function returns the number of seconds since the beginning of the day for the specified date, based on the calendar of the current context. It should not be used before the calendar was created.

Converting a date to a fraction of the current day

Synopsis:

```
FUNCTION DOUBLE_PRECISION xios_date_get_fraction_of_day(date)
TYPE(xios_date), INTENT(IN) :: date
```

Arguments:

- `date`: the date to convert

Description:

This function returns the fraction of day corresponding to the specified date based on the calendar of the current context. It should not be used before the calendar was created.