XIOS Fortran Reference Guide

Yann Meurdesoif

February 16, 2015

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 the time axis. It will appear as metadata attached to the time axis in the 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 behaviour 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 behaviour 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 Axis attribute reference

name: *string*

Fortran:

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

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

standard_name: *string*

Fortran:

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

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

long_name: *string*

Fortran:

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

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

unit: *string*

Fortran:

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

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

size: *integer*

Fortran:

```
INTEGER :: size
```

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

zoom_begin: *integer*

Fortran:

```
INTEGER :: zoom_begin
```

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

zoom_end: *integer*

Fortran:

```
INTEGER :: zoom_end
```

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

zoom_size: *integer*

Fortran:

```
INTEGER :: zoom_size
```

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

value: *1D-array of double*

Fortran:

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

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

1.4 Domain attribute reference

name: *string*

Fortran:

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

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

type: *enumeration { regular, curvilinear, unstructured }*

Fortran:

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

Define the type of the grid.

ni_glo: *integer*

Fortran:

```
INTEGER :: ni_glo
```

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

nj_glo: *integer*

Fortran:

```
INTEGER :: nj_glo
```

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

ni: *integer*

Fortran:

```
INTEGER :: ni
```

Define the first dimension of the local domain. This attribute may be also computed from **ibegin** and **iend** attribute value, so this attribute is optional. But, if defined, a consistent value must be supplied.
$ni = iend - ibegin + 1$
$1 \leq ibegin \leq iend \leq ni\_glo$

ibegin: *integer*

Fortran:

```
INTEGER :: ibegin
```

Define the begining index of the first dimension of the local domain. This attribute may be also computed from **ni** and **iend** attribute values, so this attribute is optional. But, if defined, a consistent value must be supplied.
$ibegin = iend - ni + 1$
$1 \leq ibegin \leq iend \leq ni\_glo$

iend: *integer*

Fortran:

```
INTEGER :: iend
```

Define the end index of the first dimension of the local domain. This attribute may be also computed from **ni** and **ibegin** attribute values, so this attribute is optional. But, if defined, a consistent value must be supplied.
$iend = ibegin + ni - 1$
$1 \leq ibegin \leq iend \leq ni\_glo$

nj: *integer*

Fortran:

```
INTEGER :: nj
```

Define the second dimension of the local domain. This attribute may be also computed from **jbegin** and **jend** attribute values, so this attribute is optional. But, if defined, a consistent value must be supplied.
$nj = jend - jbegin + 1$
$1 \leq jbegin \leq jend \leq nj\_glo$

jbegin: *integer*

Fortran:

```
INTEGER :: jbegin
```

Define the beginning index of the second dimension of the local domain. This attribute may be also computed from **nj** and **jend** attribute values, so this attribute is optional. But, if defined, a consistent value must be supplied.

$jbegin = jend - nj + 1$

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

jend: *integer*

Fortran:

```
INTEGER :: jend
```

Define the ending index of the second dimension of the local domain. This attribute may be also computed from **nj** and **jbegin** attribute values, so this attribute is optional. But, if defined, a consistent value must be supplied.

$jend = jbegin + nj - 1$

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

zoom_ni: *integer*

Fortran:

```
INTEGER :: zoom_ni
```

Define the size of the zoom on the first dimension on the global domain. This attribute is optional. The default value is **ni**.

$1 \leq zoom\_ni \leq ni$

zoom_ibegin: *integer*

Fortran

```
INTEGER :: zoom_ibegin
```

Define the beginning index on the first dimension of the zoom for the global domain. This attribute is optional. The default value is**1**.

zoom_nj: *integer*

Fortran:

```
INTEGER :: zoom_nj
```

Define the size of the zoom on the second dimension for the global domain. This attribute is optional. The default value is**nj**.

$1 \leq zoom\_ni \leq ni$

zoom_jbegin: *integer*

Fortran:

```
INTEGER :: zoom_jbegin
```

Define the beginning index on the second dimension of the zoom for the global domain. This attribute is optional. The default value is **1**.

mask: *2D-array of bool*

Fortran:

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

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

lonvalue: *1D-array of double*

Fortran:

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

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

latvalue: *1D-array of double*

Fortran:

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

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

nvertex: *integer*

Fortran:

```
INTEGER :: nvertex
```

Define the the maximum number of vertices for a cell. This is useful to specify the boundaries of cells for an unstructured mesh. This attribute is optional.

bounds_lon: *1D-array of double*

Fortran:

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

Longitude value of the vertex of the cells.**nvertex** attribute must also be defined. This attribute is optional.

bounds_lat: *1D-array of double*

Fortran:

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

Latitude value of the vertex of the cells.**nvertex** attribute must also be defined. This attribute is optional.

data_dim: *integer*

Fortran:

```
INTEGER :: datadim
```

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

data_ibegin: *integer*

Fortran:

```
INTEGER :: data_ibegin
```

Define the beginning index of the field data for the first dimension. This attribute is an offset regarding the local domain, so the value can be negative. A negative value 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. This attribute is optional and the default value is 0. Otherwise **data_ibegin** and **data_ni** must be defined together.

data_ni: *integer*

Fortran:

```
INTEGER :: data_ni
```

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

data_jbegin: *integer*

Fortran:

```
INTEGER :: data_jbegin
```

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

data_nj: *integer*

Fortran:

```
INTEGER :: data_nj
```

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

data_n_index: *integer*

Fortran:

```
INTEGER :: data_nindex
```

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

data_i_index: *1D-array of integer*

Fortran:

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

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

data_j_index: *1D-array of integer*

Fortran:

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

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

1.5 Grid attribute reference

name: string

Fortran:

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

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

domain_ref: string

Fortran:

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

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

axis_ref: string

Fortran:

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

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

mask: *3D-array of bool*

Fortran:

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

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

1.6 Field attribute reference

name: *string*

Fortran:

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

Define the **name** of the field as it will appear in an output file. This attribute is optional. If not present, the identifier **id** will be substituted.

standard_name: *string*

Fortran:

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

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

long_name: *string*

Fortran:

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

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

unit: *string*

Fortran:

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

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

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

Fortran:

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

Define the temporal operation applied on the field. This attribute is optional, by default no operation is applied.

freq_op: *duration*

Fortran:

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

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

freq_offset: *duration*

Fortran:

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

Define the offset when **freq_op** is defined. This attribute is optional, the default value is **0ts**(0 time step).

$0 \leq freq\_offset < freq\_op$

level: *integer*

Fortran:

```
INTEGER :: level
```

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

prec: *integer*

Fortran:

```
INTEGER :: prec
```

Define the precision in byte of a field in an output file. Available value are: 2 (integer), 4 (float single precision) and 8 (float double precision).

enabled: *bool*

Fortran:

```
LOGICAL :: enabled
```

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

field_ref: *string*

Fortran:

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

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

grid_ref: *string*

Fortran:

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

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

domain_ref: *string*

Fortran:

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

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

axis_ref: *string*

Fortran:

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

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

default_value: *double*

Fortran:

```
DOUBLE PRECISION :: default_value
```

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

valid_min: *double*

Fortran:

```
DOUBLE PRECISION :: valid_min
```

All field values below **valid_min** attribute value are set to missing value.

valid_min: *double*

Fortran:

```
DOUBLE PRECISION :: valid_min
```

All field values above **valid_max** attribute value are set to missing value.

detect_missing_value: *bool*

Fortran:

```
LOGICAL: detect_missing_value
```

When XIOS detect a default value in a field, it does not include the value in the statistic of the operation, like averaging, minimum, maximum...

add_offset: *double*

Fortran:

```
DOUBLE PRECISION: add_offset
```

Set the **add_offset** metadata CF attribute in the output file. In output, the **add_offset** value is subtracted to the field values.

scale_factor: *double*

Fortran:

```
DOUBLE PRECISION: scale_factor
```

Set the **scale_factor** metadata CF attribute in the output file. In output, the field values are divided by the **scale_factor** value.

1.7 Variable attribute reference

name: *string*

Fortran:

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

Define the **name** of the variable as it will appear in an output file. This attribute is optional.

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

Fortran:

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

Define the **type** of the variable. Note that the ***int*** type is a synonym for ***int32***. This attribute is optional.

1.8 File attribute reference

name: *string*

Fortran:

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

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

name_suffix: *string*

Fortran:

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

Define a suffix to add to the name of the output file. This attribute is optional.

min_digits: *integer*

Fortran:

```
INTEGER :: min_digits
```

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

output_freq: *duration*

Fortran:

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

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

output_level: *integer*

Fortran:

```
INTEGER :: output_level
```

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

sync_freq: *duration*

Fortran:

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

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

split_freq: *duration*

Fortran:

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

Define the time frequency for splitting the current file. In that case, the start and end dates are added to the file **name** (see **split_freq_format** attribute). This attribute is optional, by default no splitting is done.

split_freq_format: *string*

Fortran:

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

Define the format of the split date suffixed to the file. Can contain any character, *%y* will be replaced by the year (4 characters), `%mo` 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. This attribute is optional and the default behavior is to create a suffix with the date until the smaller non zero unit. For example, in one day split frequency, the hour, minute and second will not appear in the suffix, only year, month and day.

enabled: *bool*

Fortran:

```
LOGICAL :: enabled
```

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

type: *enumeration { one_file, multiple_file}*

Fortran:

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

Define the type of the output file: ***multiple_file***: one file by server using sequential netcdf writing, ***one_file***: one single global file is wrote using netcdf4 parallel access. This attribute is mandatory.

format: *enumeration { netcdf4, netcdf4_classic}*

Fortran:

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

Define the format of the output file: ***netcdf4***: the HDF5 format will be used, ***netcdf4_classic***: the classic NetCDF format will be used. The attribute is optional, 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: *enumeration { collective, independent }*

Fortran:

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

For parallel writing, define which type of MPI calls will be used. This attribute is optional, the default value is ***collective***.

append: *bool*

Fortran:

```
LOGICAL :: append
```

Define whether the output data is to be appended at the end of the file if it already exists or if the existing file is to be overwritten. This attribute is optional, the default value is ***false***.

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 enable the models to interact, complete or query the XML tree data base. New elements or group of elements can be added as child in the tree, attributes of the elements can be set or query. The type of element actually 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 element or group of element.

Arguments:

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

Description:

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

Query for a valid element

Synopsis:

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

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

Arguments:

- `id`: string identifier.

Description:

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

Adding child

Synopsis:

```
SUBROUTINE xios_add_element(parent_handle, child_handle, child_id)
TYPE(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 element or group of element.

Arguments:

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

Description:

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

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

Synopsis:

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

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

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

Arguments:

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

Description:

This subroutine my be used to query if one or more attributes of an element have a defined value. The list of attributes and their type are described in a specific chapter of the 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 element or group of element. attribute_x is describing in the chapter dedicated to the attribute description.

Arguments:

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

Description:

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

Arguments:

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

Description:

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

Arguments:

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

Description:

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

Arguments:

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

Description:

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

Arguments:

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

Description:

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

Interface relative to context management

XIOS context initialization

Synopsis:

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

Argument:

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

Description:

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

XIOS context finalization

Synopsis:

```
SUBROUTINE xios_context_finalize()
```

Arguments:

None

Description:

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

Setting current active context

Synopsis:

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

Closing definition

Synopsis:

```
SUBROUTINE xios_close_context_definition()
```

Arguments:

None

Description:

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

Interface relative to calendar management

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.

Interface relative to duration handling

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 must 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 must not be used before the calendar was created.