**C-Coupler2** **User Guide**

***Edited by:***

**Li Liu, Cheng Zhang, Ruizhe Li**

21st, January, 2015

**Copyright Notice**Copyright 2017 by Tsinghua University

All rights reserved.

No parts of this document should be either reproduced or commercially used without prior agreement by Tsinghua University representatives.

**How to get assistance?**

Assistance can be obtained as listed below

**Electronic Mail Addresses**

|  |  |  |
| --- | --- | --- |
| **Name** | **Affiliation** | **e-mail** |
| Li Liu  Cheng Zhang  Ruizhe Li | DESS,  Tsinghua University | liuli-cess@tsinghua.edu.cn |

Content

[**C-Coupler2 User Guide** 1](#_Toc493416244)

[1 Introduction 1](#_Toc493416245)

[2 Interfacing a component model with C-Coupler2 APIs 2](#_Toc493416246)

[2.1 Overall introduction 2](#_Toc493416247)

[2.2 Module to use in the code 2](#_Toc493416248)

[2.3 APIs for component model management 2](#_Toc493416249)

[2.3.1 CCPL\_register\_component 4](#_Toc493416250)

[2.3.2 CCPL\_get\_component\_id 5](#_Toc493416251)

[2.3.3 CCPL\_get\_current\_process\_id\_in\_component 5](#_Toc493416252)

[2.3.4 CCPL\_get\_component\_process\_global\_id 6](#_Toc493416253)

[2.3.5 CCPL\_get\_num\_process\_in\_component 6](#_Toc493416254)

[2.3.6 CCPL\_get\_local\_comp\_full\_name 6](#_Toc493416255)

[2.3.7 CCPL\_is\_current\_process\_in\_component 7](#_Toc493416256)

[2.3.8 CCPL\_end\_coupling\_configuration 7](#_Toc493416257)

[2.3.9 CCPL\_finalize 7](#_Toc493416258)

[2.4 API for model time management 8](#_Toc493416259)

[2.4.1 CCPL\_set\_time\_step 9](#_Toc493416260)

[2.4.2 FUNCTION CCPL\_get\_time\_step 10](#_Toc493416261)

[2.4.3 CCPL\_advance\_time 10](#_Toc493416262)

[2.4.4 CCPL\_get\_number\_of\_current\_step 10](#_Toc493416263)

[2.4.5 CCPL\_get\_number\_of\_total\_steps 11](#_Toc493416264)

[2.4.6 CCPL\_get\_current\_date 11](#_Toc493416265)

[2.4.7 CCPL\_get\_current\_year 11](#_Toc493416266)

[2.4.8 CCPL\_get\_current\_num\_days\_in\_year 12](#_Toc493416267)

[2.4.9 CCPL\_get\_current\_second 12](#_Toc493416268)

[2.4.10 CCPL\_get\_start\_time 12](#_Toc493416269)

[2.4.11 CCPL\_get\_stop\_time 13](#_Toc493416270)

[2.4.12 CCPL\_get\_previous\_time 13](#_Toc493416271)

[2.4.13 CCPL\_get\_current\_time 13](#_Toc493416272)

[2.4.14 CCPL\_is\_first\_step 14](#_Toc493416273)

[2.4.15 CCPL\_is\_first\_restart\_step 14](#_Toc493416274)

[2.4.16 CCPL\_get\_num\_elapsed\_days\_from\_start 14](#_Toc493416275)

[2.4.17 CCPL\_get\_num\_elapsed\_days\_from\_reference 15](#_Toc493416276)

[2.4.18 CCPL\_is\_end\_current\_day 15](#_Toc493416277)

[2.4.19 CCPL\_is\_end\_current\_month 15](#_Toc493416278)

[2.4.20 CCPL\_get\_current\_calendar\_time 16](#_Toc493416279)

[2.4.21 CCPL\_check\_current\_time 16](#_Toc493416280)

[2.4.22 CCPL\_is\_model\_run\_ended 16](#_Toc493416281)

[2.4.23 CCPL\_define\_single\_timer 17](#_Toc493416282)

[2.4.24 CCPL\_define\_complex\_timer 18](#_Toc493416283)

[2.4.25 CCPL\_is\_timer\_on 18](#_Toc493416284)

[2.5 API for grid management 18](#_Toc493416285)

[2.5.1 CCPL\_register\_H2D\_grid\_via\_global\_data 20](#_Toc493416286)

[2.5.2 CCPL\_register\_H2D\_grid\_via\_local\_data 22](#_Toc493416287)

[2.5.3 CCPL\_register\_H2D\_grid\_via\_file 24](#_Toc493416288)

[2.5.4 Function CCPL\_register\_H2D\_grid\_from\_another\_component 27](#_Toc493416290)

[2.5.5 CCPL\_register\_V1D\_Z\_grid\_via\_model\_data 27](#_Toc493416291)

[2.5.6 CCPL\_register\_V1D\_SIGMA\_grid\_via\_model\_data 28](#_Toc493416292)

[2.5.7 CCPL\_register\_V1D\_HYBRID\_grid\_via\_model\_data 29](#_Toc493416293)

[2.5.8 CCPL\_register\_MD\_grid\_via\_multi\_grids 29](#_Toc493416294)

[2.5.9 CCPL\_register\_mid\_point\_grid 30](#_Toc493416295)

[2.5.10 CCPL\_set\_3D\_grid\_dynamic\_surface\_field 30](#_Toc493416296)

[2.5.11 CCPL\_set\_3D\_grid\_static\_surface\_field 31](#_Toc493416297)

[2.5.12 CCPL\_set\_3D\_grid\_external\_surface\_field 31](#_Toc493416298)

[2.5.13 CCPL\_get\_grid\_size 32](#_Toc493416299)

[2.5.14 CCPL\_get\_grid\_id 32](#_Toc493416300)

[2.5.15 CCPL\_get\_H2D\_grid\_data 32](#_Toc493416301)

[2.6 API for parallel decomposition management 33](#_Toc493416302)

[2.6.1 CCPL\_register\_parallel\_decomp 34](#_Toc493416303)

[2.7 API for coupling field management 34](#_Toc493416304)

[2.7.1 CCPL\_register\_field\_instance 35](#_Toc493416305)

[2.7.2 CCPL\_register\_IO\_field\_from\_field\_instance 36](#_Toc493416306)

[2.7.3 CCPL\_register\_IO\_fields\_from\_field\_instances 36](#_Toc493416307)

[2.7.4 SUBROUTINE CCPL\_register\_IO\_field\_from\_data\_buffer 37](#_Toc493416308)

[2.8 API for coupling interface management 37](#_Toc493416309)

[2.8.1 CCPL\_register\_export\_interface 39](#_Toc493416310)

[2.8.2 CCPL\_register\_import\_interface 40](#_Toc493416311)

[2.8.3 CCPL\_register\_normal\_remap\_interface 41](#_Toc493416312)

[2.8.4 CCPL\_register\_frac\_based\_remap\_interface 42](#_Toc493416313)

[2.8.5 CCPL\_get\_comp\_full\_name\_via\_interface\_tag 43](#_Toc493416314)

[2.8.6 CCPL\_connect\_fixed\_interfaces 43](#_Toc493416315)

[2.8.7 CCPL\_execute\_interface\_using\_id 44](#_Toc493416316)

[2.8.8 CCPL\_execute\_interface\_using\_name 45](#_Toc493416317)

[2.9 API for restart management 46](#_Toc493416318)

[2.9.1 CCPL\_do\_restart\_write 46](#_Toc493416319)

[2.9.2 CCPL\_do\_restart\_read 47](#_Toc493416320)

[2.9.3 CCPL\_is\_restart\_timer\_on 47](#_Toc493416321)

[2.10 API for parallel debugging 47](#_Toc493416322)

[2.10.1 CCPL\_report\_log 47](#_Toc493416323)

[2.10.2 CCPL\_report\_progress 48](#_Toc493416324)

[2.10.3 CCPL\_report\_error 48](#_Toc493416325)

[2.11 Examples of constructing a coupled system with C-Coupler2 APIs 48](#_Toc493416326)

[3 Coupling configuration with configuration files 53](#_Toc493416327)

[3.1 env\_run.xml 54](#_Toc493416328)

[3.2 public\_field\_attribute.xml 56](#_Toc493416329)

[3.3 CCPL\_report.xml 57](#_Toc493416330)

[3.4 Remapping configuration 58](#_Toc493416331)

[3.4.1 Remapping weight file 58](#_Toc493416332)

[3.4.2 Configuration file “\*remapping\_configuration.xml” 60](#_Toc493416333)

[3.4.3 A priority based hierarchical scheme 63](#_Toc493416334)

[3.5 *Comp\_full\_name*.import.redirection.xml 64](#_Toc493416335)

[3.5.1 XML node of “component\_import\_interfaces\_configuration” 64](#_Toc493416336)

[3.5.2 XML node of “component\_grid\_redirection\_configuration” 67](#_Toc493416337)

[4 Outputs from C-Coupler2 68](#_Toc493416338)

## Introduction

C-Coupler2 is a new coupler targeting for wide usage in constructing various models for predicting or simulating the weather, wave, and climate. It is the second version in the C-Coupler family, which is mainly developed and maintained by the C-Coupler team in Tsinghua University, China.

## Coupling configuration with C-Coupler2 APIs

## Overall introduction

C-Coupler2 works as a shared library for model coupling. C-Coupler2 provides near 70 Fortran APIs which enable the component models in a coupled system to flexibly describe various coupling configurations. To enhance the software debugging capability regarding to C-Coupler2, the last parameter in each API is an optional input string named “annotation”. When “annotation” is given, C-Coupler2 will include it into an error report, which will enable users to easily locate the model code corresponding to an error report.

The C-Coupler2 APIs can be classified into the following types:

1. APIs for component model management
2. APIs for model time management
3. APIs for grid management
4. APIs for parallel decomposition management
5. APIs for coupling field management
6. APIs for coupling interface management
7. APIs for restart management
8. APIs for parallel debugging

## Module to use in the code

## APIs for component model management

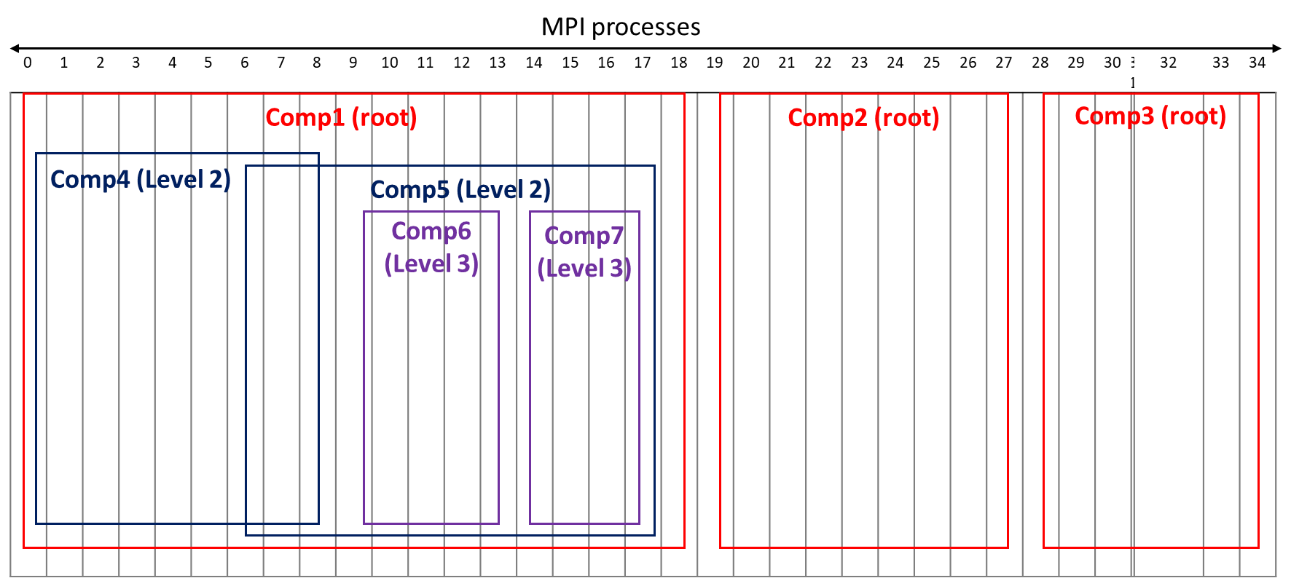


Figure X An example of the layout of processes among component models

C-Coupler2 only serves the component models that have been registered to it. When registering a component model, its name and type are required to be specified. Component models can be registered recursively, which means that a component model can be the parent of a set of component models. A component model that does not have the parent is called a root component model. C-Coupler2 will allocate a unique ID and generate a unique full name for each component model. The full name of a component model is formatted as “*parent\_full\_name*@*model\_name*”, where “*model\_name*” means the name of the component model and “*parent\_full\_name*” means the full name of its parent. A component model is either active or pseudo (inactive), which is specified through the model type. An active component model can be used to specify coupling configurations and its name will be used to generate the full name of its children models, while users cannot specify the coupling configurations corresponding to a pseudo component model whose name will not be included in the full name of any component model. Table X lists out the model types that are currently supported by C-Coupler2. Please note the model types of “active\_coupled\_system” and “pseudo\_coupled\_system”, which indicates that a component model registered to C-Coupler2 can be a coupled model.

Each component model corresponds to an MPI communicator that includes a set of processes (or MPI tasks). As C-Coupler2 can handle model coupling among executables, intra the same executable or even intra the same process, it can support various process layouts among component models. C-Coupler2 only introduces two constraints in process layout: (1) each process in the communicator MPI\_COMM\_WORLD must belong to a unique root process (for example, “comp1”, “comp2” and “comp3” in Figure X are root components and each process belongs to only one component); a component model must include all processes of its children (for example, in Figure X, “comp6” and “comp7” are children of “comp5”, and “comp5” includes all processes of “comp6” and “comp7”). C-Coupler2 enables two non-root component models to share the same processes (for example, in Figure X, “comp4” and “comp5”, which are the children of “comp1”, share some processes), and enables a component model to be a coupled system that that consists of the component models corresponding to one executable or even multiple executables .

Table X lists out the APIs for component model management as well as their brief descriptions.

Table X Model types that are currently supported by C-Coupler2

|  |  |  |
| --- | --- | --- |
| Model type | Description | Remark |
| cpl | Coupler | Active component model |
| atm | Atmosphere model | Active component model |
| glc | Glacier model | Active component model |
| atm\_chem | Atmospheric chemistry model | Active component model |
| ocn | Ocean model | Active component model |
| lnd | Land surface model | Active component model |
| sea\_ice | Sea ice model | Active component model |
| wave | Wave model | Active component model |
| roff | Runoff model | Active component model |
| active\_coupled\_system | Coupled model that consists of a set of component models | Active component model |
| pseudo\_coupled\_system | Coupled model that consists of a set of component models | Pseudo component model |

Table X APIs for component model management

|  |  |  |
| --- | --- | --- |
| No. | API | Brief description |
| 1 | CCPL\_register\_component | Register a component model to C-Coupler |
| 2 | CCPL\_get\_component\_id | Get the ID of a component model corresponding to the given model name |
| 3 | CCPL\_get\_current\_process\_id\_in\_component | Get the ID of the current process in the given component model |
| 4 | CCPL\_get\_component\_process\_global\_id | Get the global ID of a process in the given component model |
| 5 | CCPL\_get\_num\_process\_in\_component | Get the total number of processes in the MPI communicator of the given component model |
| 6 | CCPL\_get\_local\_comp\_full\_name | Get the full name of the given component model |
| 7 | CCPL\_is\_current\_process\_in\_component | Check whether the current process is in the given component model |
| 8 | CCPL\_end\_coupling\_configuration | Finalize the coupling configuration stage of the given component model |
| 9 | CCPL\_finalize | Finalize the model coupling by C-Coupler2 |

## CCPL\_register\_component

* **integer FUNCTION CCPL\_register\_component(parent\_id, comp\_name, comp\_type, comp\_comm, annotation)**
* return value [INTEGER; OUT]: The ID of the new component model
* parent\_id [INTEGER; IN]: The ID of the parent component model. When the new component model does not have the parent, which means that the new component model is the root component model, “parent\_id” should be -1.
* comp\_name [CHARACTER; IN]: The name of the new component model. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘\_’.
* comp\_type [CHARACTER; IN]: The type of the new component mode. The model types that are currently supported by C-Coupler2 are shown in Table X.
* comp\_comm [INTEGER; INOUT]: The MPI communicator corresponding to the new component model. “comp\_comm” can be specified as *-1*, which means the MPI communicator corresponding to the new component model is unknown and then will be created by C-Coupler2 and then returned through “comp\_comm”; “comp\_comm” can also be a known MPI communicator that has already been created beyond C-Coupler2. When “comp\_comm” is not -1, users must guarantee that “comp\_comm” is a legal MPI communicator.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API registers a new component model to C-Coupler2 and returns its ID when the registration succeeds. For each process, the first component model to be registered must be the root component and only one root component can be registered. If “comp\_comm” is -1, all processes in the parent component model corresponding to “parent\_id” must call this API at the same time (when “parent\_id” is -1, all processes in the communicator MPI\_COMM\_WORLD must call this API at the same time), and “comp\_name” is the key for a component model (the processes with the same “comp\_name” will be classified into the same component model). If “comp\_comm” is a valid MPI communicator, all processes in this MPI communicator must call this API at the same time with the same “comp\_name” and “comp\_type”. Any two children components of the same component or any two root components cannot share the same “comp\_name”. All component models registered by the same process cannot share the same “comp\_name”.

This API will start the stage of coupling configuration of the corresponding component model.

## CCPL\_get\_component\_id

* **integer FUNCTION CCPL\_get\_component\_id(comp\_name, annotation)**
* Return value [INTEGER; OUT]: The ID of the component model; a return value of *-1* means that no component model with the given “comp\_name” is found.
* comp\_name [CHARACTER; IN]: The name of a component model. It has a maximum length of 80 characters.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API returns the ID of a component model that has been registered in the current process, according to the given model name.

## CCPL\_get\_current\_process\_id\_in\_component

* **integer FUNCTION CCPL\_get\_current\_process\_id\_in\_component(comp\_id, annotation)**
* Return value [INTEGER; OUT]: The ID of the current process in the MPI communicator of the given component model.
* comp\_id [INTEGER; IN]: The ID of the given component model
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API returns the ID of the current process in the MPI communicator of the given component model that should have already been registered in the current process.

## CCPL\_get\_component\_process\_global\_id

* **integer FUNCTION CCPL\_get\_component\_process\_global\_id(comp\_id, local\_proc\_id, annotation)**
* Return value [INTEGER; OUT]: The global ID of the given process in the given component model.
* comp\_id [INTEGER; IN]: The ID of the given component model
* local\_proc\_id [INTEGER; IN]: The local ID of the given process in the given component model. It should be no smaller than 0 and smaller than the total number of processes in the given component model.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API returns the global ID of a given process in the given component model that should have already been registered in the current process.

## CCPL\_get\_num\_process\_in\_component

* **integer FUNCTION CCPL\_get\_num\_process\_in\_component(comp\_id, annotation)**
* Return value [INTEGER; OUT]: The total number of processes in the given component model.
* comp\_id [INTEGER; IN]: The ID of the given component model
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API returns the total number of processes in the given component model that should have already been registered in the current process.

## CCPL\_get\_local\_comp\_full\_name

* **SUBROUTINE CCPL\_get\_local\_comp\_full\_name(comp\_id, comp\_full\_name, annotation)**
* comp\_id [INTEGER; IN]: The ID of the given component model
* comp\_full\_name [CHARACTER; OUT]: The full name of the given component model. This string should have sufficient length, for example with a length of 512 characters.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API gets the full name of the given component model that should have already been registered in the current process.

## CCPL\_is\_current\_process\_in\_component

* **logical FUNCTION CCPL\_is\_current\_process\_in\_component(comp\_full\_name, annotation)**
* Return value [LOGICAL; OUT]: If the current process is in the given component model, “true” will be returned; otherwise “false” will be returned.
* comp\_full\_name [CHARACTER; IN]: The full name of the given component model. It has a maximum length of 512 characters.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API checks whether the current process is in the given component model that is specified through the full name.

## CCPL\_end\_coupling\_configuration

* **SUBROUTINE CCPL\_end\_coupling\_configuration(comp\_id, annotation)**
* comp\_id [INTEGER; IN]: The ID of the given component model
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API finalizes the stage of coupling configuration of the given component model that should have already been registered in the current process. A component model can successfully call this API only when all its children have already called this API. After calling this API, the corresponding component model cannot further register children component models, timers, grids, parallel decompositions, coupling fields, coupling interfaces, etc.. All processes of the given component model are required to call this API at the same time. At the time when all root components call this API to finalize the coupling configuration stage of all component models, the coupling generator of C-Coupler2 will be invoked to automatically generate the coupling procedures for the whole coupled system.

## CCPL\_finalize

* **SUBROUTINE CCPL\_finalize(to\_finalize\_MPI, annotation)**
* to\_finalize\_MPI [LOGICAL; IN]: when “to\_finalize\_MPI” is set to “true” and MPI has not been finalized, MPI will be finalized. C-Coupler2 will not finalize MPI when “to\_finalize\_MPI” is set to “false”.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API releases all data structures of C-Coupler2 and then finalize C-Coupler2. It can also finalize MPI at the same time when “to\_finalize\_MPI” is set to “true”. All processes in the communicator MPI\_COMM\_WORLD must call this API at the same time. For each process, CCPL\_finalize can be called only once.

## API for model time management

A model generally requires to manage its own model time information, in order to control the time integration. C-Coupler2 also manages model time information, in order to handling model coupling in time integration of the whole coupling system, with a separate and unique time manager for each active component model. Thus detailed time information of a model can also be accessed through C-Coupler2 and a model can employ C-Coupler2 for its model time management. For an active component model that has its own model time management, it should be guaranteed that the model time is constantly consistent between the component model and C-Coupler2. The API “CCPL\_check\_current\_time” can be used to check such consistency.

The unique time manager of an active component model is not activated until its unique time step has been set. After a time manager is activated, users can access detailed information of the model time, and can define and use timers to control model coupling.

Besides managing the time information of each component model, the time managers of C-Coupler2 also share some identical global information of a simulation run (specified in the XML file “CCPL\_dir/config/all/env\_run.xml” (p)), including the case name, case description, run type (“initial”, “continue”, “branch” and “hybrid” are the four run types currently supported in C-Coupler2), start time, stop time (the simulation run can be set to endless), and the frequency to write restart files. Please note that C-Coupler2 requires all component models in a coupled system to use the same case name, case description, run type, start time, stop time and restart writing frequency.

Currently, C-Coupler2 only supports the model years from 0 to 9999. When users want C-Coupler2 to support the model years beyond this range, please contact the authors.

Table X lists out the APIs for time management as well as their brief descriptions.

Table X APIs for time management.

|  |  |  |
| --- | --- | --- |
| No. | API | Brief description |
| 1 | CCPL\_set\_time\_step | Set the unique time step of the given component model |
| 2 | CCPL\_get\_time\_step | Get the time step of the given component model, which is set by “CCPL\_set\_time\_step” |
| 3 | CCPL\_advance\_time | Advance the model time of the given component model by one time step |
| 4 | CCPL\_get\_number\_of\_current\_step | Get the number of current time step of the given component model |
| 5 | CCPL\_get\_number\_of\_total\_steps | Get the total number of time steps during the whole execution of the given component model |
| 6 | CCPL\_get\_current\_date | Get the current date of the given component model |
| 7 | CCPL\_get\_current\_year | Get the current year of the given component model |
| 8 | CCPL\_get\_current\_num\_days\_in\_year | Get the current number of days in the current year of the given component model |
| 9 | CCPL\_get\_current\_second | Get the current second of the given component model |
| 10 | CCPL\_get\_start\_time | Get the start time of model run of the given component model |
| 11 | CCPL\_get\_stop\_time | Get the stop time of model run of the given component model |
| 12 | CCPL\_get\_previous\_time | Gets the time of the previous time step of the given component model |
| 13 | CCPL\_get\_current\_time | Get the current time of the given component model |
| 14 | CCPL\_is\_first\_step | Check whether the current time step is the first step in a model run |
| 15 | CCPL\_is\_first\_restart\_step | Check whether the current time step is the first step after restarting the simulation run (first restart step). Please note that the first restart step may not be the first step of the model run. |
| 16 | CCPL\_get\_num\_elapsed\_days\_from\_start | Get the number of elapsed days from the start date to the current date of the given component model |
| 17 | CCPL\_get\_num\_elapsed\_days\_from\_reference | Get the number of elapsed days from the reference date to the current date of the given component model |
| 18 | CCPL\_is\_end\_current\_day | Check whether the current time is the end of the current day for the given component model |
| 19 | CCPL\_is\_end\_current\_month | Check whether the current time is the end of the current month for the given component model |
| 20 | CCPL\_get\_current\_calendar\_time | Get the current calendar time of the given component model |
| 21 | CCPL\_check\_current\_time | Check the consistency of model time between the given component model and C-Coupler2 |
| 22 | CCPL\_is\_model\_run\_ended | Check whether the simulation run of the given component model has reached the stop time |
| 23 | CCPL\_define\_single\_timger | Define a single timer that is a periodic timer for the given component model |
| 24 | CCPL\_define\_complex\_timer | Define a complex timer that consists of a number of timers for a component model |
| 25 | CCPL\_is\_timer\_on | Check whether the given timer is on at the current time step of the corresponding component model |

## CCPL\_set\_time\_step

* **SUBROUTINE CCPL\_set\_time\_step(comp\_id, time\_step\_in\_second, annotation)**
* comp\_id [INTEGER; IN]: The ID of the given component model
* time\_step\_in\_second [INTEGER; IN]: The time step (number of seconds) of the given component model. It must be an integer value larger than 0.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API sets the time step (number of seconds) of the given component model. Currently, C-Coupler2 requires the total number of seconds of the whole simulation run to be an integral multiple of “time\_step\_in\_second”, and requires the frequency (in seconds) of writing restart data file to be an integral multiple of “time\_step\_in\_second”. All processes of the given component model are required to call this API at the same time with the same “time\_step\_in\_second”. A component model can call this API only one time. In other word, a component model can have only one time step. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

## FUNCTION CCPL\_get\_time\_step

* **integer FUNCTION CCPL\_get\_time\_step(comp\_id, annotation)**
* Return value [INTEGER; OUT]: The size of time step (number of seconds) of the given component model. When the time step of the corresponding component model has not been set, the return value will be *-1*.
* comp\_id [INTEGER; IN]: The ID of the given component model
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API returns the size of time step of the given component model.

## CCPL\_advance\_time

* **SUBROUTINE CCPL\_advance\_time(comp\_id, annotation)**
* comp\_id [INTEGER; IN]: The ID of the given component model
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API advances the time of the corresponding component model by one time step. All processes of the component model are required to call this API at the same time. This API can be called only when the coupling configuration of the whole coupled model has been ended. This API will write all I/O fields of the given component model into data files according to the timer of each I/O field.

## CCPL\_get\_number\_of\_current\_step

* **integer FUNCTION CCPL\_get\_number\_of\_current\_step(comp\_id, annotation)**
* Return value [INTEGER; OUT]: The number of the current time step in the given component model.
* comp\_id [INTEGER; IN]: The ID of the given component model
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API returns the current time step in the component model corresponding to “comp\_id”. The number of the first time step in an initial run, or hybrid run is 0. A restart run or a branch run shares the same number of time step with the corresponding initial run.

## CCPL\_get\_number\_of\_total\_steps

* **integer FUNCTION CCPL\_get\_number\_of\_total\_steps(comp\_id, annotation)**
* return value [INTEGER; OUT]: The total number of time steps from the start to the end of the simulation run.
* comp\_id [INTEGER; IN]: The ID of the given component model.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API returns the total number of time steps of the given component model from the start to the end of the simulation run. When the simulation run has been set to be endless, “-1” will be returned.

## CCPL\_get\_current\_date

* **integer FUNCTION CCPL\_get\_current\_date(comp\_id, annotation)**
* Return value [INTEGER; OUT]: The current date (YYYYMMDD) of the given component model.
* comp\_id [INTEGER; IN]: The ID of the given component model
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API returns the current date of the given component model.

## CCPL\_get\_current\_year

* **integer FUNCTION CCPL\_get\_current\_year(comp\_id, annotation)**
* Return value [INTEGER; OUT]: The current year (YYYY) of the given component model.
* comp\_id [INTEGER; IN]: The ID of the given component model
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API returns the current year of the given component model.

## CCPL\_get\_current\_num\_days\_in\_year

* **integer FUNCTION CCPL\_get\_current\_num\_days\_in\_year(comp\_id, annotation)**
* Return value [INTEGER; OUT]: The current number of days (≥1) in the current year of the given component model.
* comp\_id [INTEGER; IN]: The ID of the given component model
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API returns the current number of days in the current year of the given component model.

## CCPL\_get\_current\_second

* **integer FUNCTION CCPL\_get\_current\_second(comp\_id, annotation)**
* Return value [INTEGER; OUT]: The current number of seconds in the current day (<86400) of the given component model.
* comp\_id [INTEGER; IN]: The ID of the given component model
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API returns the current second of the given component model.

## CCPL\_get\_start\_time

* **SUBROUTINE CCPL\_get\_start\_time(comp\_id, year, month, day, second, annotation)**
* comp\_id [INTEGER; IN]: The ID of the given component model
* year [INTEGER; OUT]: The year (YYYY) of the start time of the given component model
* month [INTEGER; OUT]: The month (MM) of the start time of the given component model
* day [INTEGER; OUT]: The day (DD) of the start time of the given component model.
* second [INTEGER; OUT]: The number of second (<86400) of the start time of the given component model.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API gets the start time (year, month, day and second) of model run of the given component model.

## CCPL\_get\_stop\_time

* **SUBROUTINE CCPL\_get\_stop\_time(comp\_id, year, month, day, second, annotation)**
* comp\_id [INTEGER; IN]: The ID of the given component model
* year [INTEGER; OUT]: The year (YYYY) of the stop time of the given component model
* month [INTEGER; OUT]: The month (MM) of the stop time of the given component model
* day [INTEGER; OUT]: The day (DD) of the stop time of the given component model.
* second [INTEGER; OUT]: The number of second (<86400) of the stop time of the given component model.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API gets the stop time (year, month, day and second) of model run of the given component model. When the simulation run has been set to be endless, “-1” will be returned through each output parameter.

## CCPL\_get\_previous\_time

* **SUBROUTINE CCPL\_get\_previous\_time(comp\_id, year, month, day, second, annotation)**
* comp\_id [INTEGER; IN]: The ID of the given component model
* year [INTEGER; OUT]: The year (YYYY) of the previous time of the given component model
* month [INTEGER; OUT]: The month (MM) of the previous time of the given component model
* day [INTEGER; OUT]: The day (DD) of the previous time of the given component model.
* second [INTEGER; OUT]: The number of second (<86400) of the previous time of the given component model.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API gets the time of the previous time step of the given component model. The previous time of the start time is the start time itself.

## CCPL\_get\_current\_time

* **SUBROUTINE CCPL\_get\_current\_time(comp\_id, year, month, day, second, shift\_second, annotation)**
* comp\_id [INTEGER; IN]: The ID of the given component model
* year [INTEGER; OUT]: The year (YYYY) of the current time of the given component model
* month [INTEGER; OUT]: The month (MM) of the current time of the given component model
* day [INTEGER; OUT]: The day (DD) of the current time of the given component model.
* second [INTEGER; OUT]: The number of second (<86400) of the current time of the given component model.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API gets the current time of the given component model. An additional number of seconds can be added when getting the current time.

## CCPL\_is\_first\_step

* **logical FUNCTION CCPL\_is\_first\_step(comp\_id, annotation)**
* return value [LOGICAL; OUT]: if the current time step is the first step of a simulation run, the return value is *true*; otherwise, the return value is *false*.
* comp\_id [INTEGER; IN]: The ID of the given component model
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API checks whether the current time step is the first step of a simulation run.

## CCPL\_is\_first\_restart\_step

* **logical FUNCTION CCPL\_is\_first\_restart\_step(comp\_id, annotation)**
* Return value [LOGICAL; OUT]: if the current time step is the first step after restarting the simulation run (first restart step), the return value is true; otherwise, the return value is false.
* comp\_id [INTEGER; IN]: The ID of the given component model
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API checks whether the current time step is the first step after restarting the simulation run (first restart step). Please note that the first restart step may not be the first step.

## CCPL\_get\_num\_elapsed\_days\_from\_start

* **SUBROUTINE CCPL\_get\_num\_elapsed\_days\_from\_start(comp\_id, num\_days, current\_second, annotation)**
* comp\_id [INTEGER; IN]: The ID of the given component model
* num\_days [INTEGER; OUT]: The number of elapsed days from the start date to the current date of the given component model.
* current\_second [INTEGER; OUT]: The second in a day of the current time of the given component model.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API gets the number of elapsed days from the start date to the current date and gets the current second of the given component model.

## CCPL\_get\_num\_elapsed\_days\_from\_reference

* **SUBROUTINE CCPL\_get\_num\_elapsed\_days\_from\_reference(comp\_id, num\_days, current\_second, annotation)**
* comp\_id [INTEGER; IN]: The ID of the given component model
* num\_days [INTEGER; OUT]: The number of elapsed days from the reference date to the current date of the given component model.
* current\_second [INTEGER; OUT]: The second in a day of the current time of the given component model.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API gets the number of elapsed days from the reference date to the current date and gets the current second of the corresponding component model.

## CCPL\_is\_end\_current\_day

* **logical FUNCTION CCPL\_is\_end\_current\_day(comp\_id, annotation)**
* return value [LOGICAL; OUT]: if the current time is the end of the current day, the return value is *true*; otherwise, the return value is *false*.
* comp\_id [INTEGER; IN]: The ID of the given component model.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API checks whether the current time is the end of the current day for the given component model.

## CCPL\_is\_end\_current\_month

* **logical FUNCTION CCPL\_is\_end\_current\_month(comp\_id, annotation)**
* Return value [LOGICAL; OUT]: if the current time is the end of the current month of the given component model, the return value is *true*; otherwise, the return value is *false*.
* comp\_id [INTEGER; IN]: The ID of the given component model.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API checks whether the current time is the end of the current month for the given component model.

## CCPL\_get\_current\_calendar\_time

* **SUBROUTINE CCPL\_get\_current\_calendar\_time(comp\_id, cal\_time, shift\_second, annotation)**
* comp\_id [INTEGER; IN]: The ID of the given component model.
* cal\_time [REAL; OUT]: The calendar time of the current time. It is a non-negative floating-point value no more than the number of days in a year.
* shift\_second [INTEGER, OPTIONAL; IN]: The additional number of seconds from the current time when calculating the calendar time. It must be 0 or a positive value.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API gets the current calendar time of the given component model. A positive number of seconds can be added as a shift when calculating the calendar time.

## CCPL\_check\_current\_time

* **SUBROUTINE CCPL\_check\_current\_time(comp\_id, date, second, annotation)**
* comp\_id [INTEGER; IN]: The ID of the given component model.
* date [INTEGER; in]: the date (YYYYMMDD) provide by the given component model.
* second [INTEGER, OPTIONAL; IN]: the number of second provide by the given component model.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API checks whether the current time of the given component model is the same as the provided date and second. An error will be reported and the model run will be ended if the checking fails.

## CCPL\_is\_model\_run\_ended

* **logical FUNCTION CCPL\_is\_model\_run\_ended(comp\_id, annotation)**
* return value [LOGICAL; OUT]: if the model run of the given component model has been ended, “true” will be returned; otherwise “false” will be returned.
* comp\_id [INTEGER; IN]: The ID of the given component model.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API checks whether the simulation run of the given component model has reached the stop time.

## CCPL\_define\_single\_timer

* **integer FUNCTION CCPL\_define\_single\_timer(comp\_id, period\_unit, period\_count, local\_lag\_count, remote\_lag\_count, annotation)**
* Return value [INTEGER; OUT]: The ID of the new timer.
* comp\_id [INTEGER; IN]: The ID of the given component model.
* period\_unit [CHARACTER; IN]: The unit for specifying the period of the single timer. The unit must be “steps” (or “nsteps”), “seconds” (or “nseconds”), “days” (or “ndays”), “months” (or “nmonths”), or “years” (or “nyears”).
* period\_count [INTEGER; IN]: The count corresponding to the unit for specifying the period of the single timer.
* local\_lag\_count [INTEGER; IN]: A count corresponding to the period unit. It is used to specify a lag (can be viewed as a time offset from the start time) which will impact when the single timer is on.
* remote\_lag\_count [INTEGER, OPTIONAL; IN]: A count corresponding to the period unit. It can be used to specify a lag for a coupling connection between two component models or intra one component model. Its default value is 0, which means no lag. Please note that the lag for a coupling connection is determined by the timer of the receiver component model. The lag corresponding to a coupling connection can be viewed as a time difference from the receiver component model to the sender component model, which can control the time sequence between the two component models. For example, given a lag of 1 hour, the coupling fields produced by the sender component model at the sender’s 0th hour will be obtained by the receiver component model at the receiver’s 1st hour; given a lag of -1 hour, the coupling fields produced by the sender component model at the sender’s 1st hour will be obtained by the receiver component model at the receiver’s 0th hour. Thus, users can flexibly achieve concurrent run or sequential run between component models. Please note that, wrong setting of “remote\_lag\_count” may introduce deadlocks between component models.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API defines a single timer that is a periodic timer for the given component model. “period\_unit” and “period\_count” are used to specify the period of the single timer. In default, the single timer will be on at the start time of simulation. When users want to change the model time when the single timer is on, users can set a lag count (with positive or negative value) through the parameter “local\_lag\_count”. The period of the single timer should be consistent with the time step size of the component model, which means the period in should be an integer multiple of the time step size. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

For example, give a single timer of <period\_unit=“steps”, period\_count=“5”, local\_lag\_count=“2”>, it will be on at the 2nd, 7th (5\**i*+2, where *i* is a non-negative integer) step of the given component model.

## CCPL\_define\_complex\_timer

* **integer FUNCTION CCPL\_define\_complex\_timer(comp\_id, num\_children\_timers, children\_timers\_id, OR\_or\_AND, annotation)**
* return value [INTEGER; OUT]: The ID of the new timer.
* comp\_id [INTEGER; IN]: The ID of the given component model.
* num\_children\_timers [INTEGER; IN]: The number of children timers used to define this complex timer.
* children\_timers\_id [INTEGER, DIMENSION|(:); IN]: The ID of children timers used to define this complex timer. All children timers share the same component model corresponding to “comp\_id”. The size of “children\_timers\_id” cannot be smaller than “num\_children\_timers”.
* OR\_or\_AND [INTEGER; IN]: value of 0 means *or*: the new complex timer is on when one of its children timers is on; value of 1 means *and*: the new complex timer is on when all of its children timers are on.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API defines a complex timer that consists of a number of children timers (a children timer can be a single timer or a complex timer) for the given component model. It cannot be called after the coupling configuration stage of the given component model has been ended.

## CCPL\_is\_timer\_on

* **logical FUNCTION CCPL\_is\_timer\_on(timer\_id, annotation)**
* Return value [LOGICAL; OUT]: If the given timer is on at the current time step of the corresponding component model, *true* will be returned; otherwise *false* will be returned.
* timer\_id [INTEGER: IN]: the ID of the given timer.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API checks whether the given timer is on at the current time step of the corresponding component model.

## API for grid management

C-Coupler2 requires itself to manage the model grids related to coupling. In detail, C-Coupler can manage various kinds of horizontal grids (2-D), vertical grids (1-D) with Z, SIGMA or HYBRID coordinates, and time grids (1-D), and also can manage multi-dimension grids each of which consists of a horizontal grid, a vertical grid or a time grid. C-Coupler2 offers multiple choices for registering model grids. A component model can register a model grid to C-Coupler2 through model data or a grid data file, or from another component model.

For a 3-D grid that consists of a horizontal grid and a vertical grid with SIGMA or HYBRID coordinate, C-Coupler2 enables to set its unique surface field that is on the horizontal grid, in order for calculating the vertical coordinates at each horizontal grid point. The surface field of a 3-D grid can be static, dynamic and external. A static surface field means that its values are constant along with the time integration, so that the vertical coordinate values in the corresponding 3-D grid are constant. A dynamic surface field means that its values change along with the time integration, so that the vertical coordinate values in the corresponding 3-D grid are inconstant. An external surface field means that its values are determined by the surface field of another 3-D grid.

Most models are parallelized with MPI, where the model grids are decomposed into separate domains for parallel integration. Besides managing global model grids, C-Coupler2 also manages decomposed model grids at the same time. Model code can access the grid data in a global model grid or the grid data in decomposed model grid.

In C-Coupler2, a grid is associated with a unique active component model. For a grid that is shared by multiple component models, it should be registered to each component model separately. The keyword for a grid can be expressed as <ID of component model, grid name>. Therefore, different grids in the same component model cannot have the same grid name, while the grids in different component models can have the same grid name.

Table X lists out the APIs for grid management as well as their brief descriptions.

Table X APIs for grid management.

|  |  |  |
| --- | --- | --- |
| No. | API | Brief description |
| 1 | CCPL\_register\_H2D\_grid\_via\_global\_data | Register a horizontal grid using global grid data |
| 2 | CCPL\_register\_H2D\_grid\_via\_local\_data | Register a horizontal grid using local grid data |
| 3 | CCPL\_register\_H2D\_grid\_via\_file | Register a horizontal grid using a data file |
| 4 | CCPL\_register\_H2D\_grid\_from\_another\_component | Register a horizontal grid based on another component |
| 5 | CCPL\_register\_V1D\_Z\_grid\_via\_model\_data | Register a vertical Z grid using model data |
| 6 | CCPL\_register\_V1D\_SIGMA\_grid\_via\_model\_data | Register a vertical SIGMA grid using model data |
| 7 | CCPL\_register\_V1D\_HYBRID\_grid\_via\_model\_data | Register a vertical HYBRID grid using model data |
| 8 | CCPL\_register\_MD\_grid\_via\_multi\_grids | Register a grid using multiple registered grids |
| 9 | CCPL\_register\_mid\_point\_grid | Register of the middle-point grid of a 3-D interface-level grid |
| 10 | CCPL\_set\_3D\_grid\_dynamic\_surface\_field | Set the dynamic surface field of a 3-D grid |
| 11 | CCPL\_set\_3D\_grid\_static\_surface\_field | Set the static surface field of a 3-D grid |
| 12 | CCPL\_set\_3D\_grid\_external\_surface\_field | Declare that the surface field of a 3-D grid is external |
| 13 | CCPL\_get\_grid\_size | Get the global size of a grid |
| 14 | CCPL\_get\_grid\_id | Get the ID of a grid |
| 15 | CCPL\_get\_H2D\_grid\_data | Get grid data of a horizontal grid |

## CCPL\_register\_H2D\_grid\_via\_global\_data

* **integer FUNCTION CCPL\_register\_H2D\_grid\_via\_global\_data(comp\_id, grid\_name, edge\_type, coord\_unit, cyclic\_or\_acyclic, dim\_size1, dim\_size2, min\_lon, max\_lon, min\_lat, max\_lat, center\_lon, center\_lat, mask, area, vertex\_lon, vertex\_lat, annotation)**
* return value [INTEGER; OUT]: The ID of the new horizontal (H2D) grid.
* comp\_id [INTEGER; IN]: The ID of the given component model that the new grid is associated with.
* grid\_name [CHARACTER; IN]: The name of the new grid. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘\_’.
* edge\_type [CHARACTER; IN]: The type of the edges of grid cells (“LON\_LAT”， “XY”, “GREAT\_ARC”, or “TriPolar”).
* coord\_unit [CHARACTER; IN]: The unit of the coordinates of the new grid. When the edge type is “LON\_LAT”, “GREAT\_ARC” or “TriPolar”, the unit must be “degrees” or “radians”.
* cyclic\_or\_acyclic [CHARACTER; IN]: Whether the longitude (X) dimension is cyclic or acyclic. The value must be “cyclic” or “acyclic”.
* dim\_size1 [INTEGER; IN]: The size of the size of longitude (X) dimension or of the global grid. “dim\_size1” must be larger than 3.
* dim\_size2 [INTEGER; IN]: If “dim\_size1” has been set to the size of the global grid, “dim\_size2” must be set to 0; otherwise, “dim\_size2” must be set to the size of latitude (Y) dimension. “dim\_size2” must be 0 or larger than 3.
* min\_lon [REAL; IN]: Minimum longitude value of the boundary of the domain of the global grid. Its value can be a specific value between *-360.0* and *360.0* degrees (or between *-2PI* and *2PI*) or be *-999.0*. The value of *-999.0* indicates that the component model wants C-Coupler2 to calculate “min\_lon” automatically. Please note that, “min\_lon” calculated by C-Coupler2 may be incorrect.
* max\_lon [REAL; IN]: Maximum longitude value of the boundary of the domain of the global grid. Its value can be a specific value between *-360.0* and *360.0* degrees (or between *-2PI* and *2PI*) or be *-999.0*. The value of *-999.0* indicates that the component model wants C-Coupler2 to calculate “max\_lon” automatically. Please note that, “max\_lon” calculated by C-Coupler2 may be incorrect. “min\_lon” and “max\_lon” must be *-999.0* or not *-999.0* at the same time. When both “min\_lon” and “max\_lon” are not *-999.0*, “min\_lon” can be larger or smaller than “max\_lon”.
* min\_lat [REAL; IN]: Minimum latitude value of the boundary of the domain of the global grid. Its value can be a specific value between *-90.0* and *90.0* degrees (or between *–PI/2* and *PI/2*) or be *-999.0*. The value of *-999.0* indicates that the component model wants C-Coupler2 to calculate “min\_lat” automatically. Please note that, “min\_lat” calculated by C-Coupler2 may be incorrect. When “min\_lat” has been set to *-90* degrees (or *–PI/2*), it indicates that the global grid covers the South Pole.
* max\_lat [REAL; IN]: Maximum latitude value of the boundary of the domain of the global grid. Its value can be a specific value between *-90.0* and *90.0* degrees (or between *–PI/2* and *PI/2*) or be *-999.0*. The value of *-999.0* indicates that the component model wants C-Coupler2 to calculate “max\_lat” automatically. Please note that, “max\_lat” calculated by C-Coupler2 may be incorrect. When “max\_lat” has been set to *90* degrees (or *PI/2*), it indicates that the global grid covers the North Pole. When both “min\_lat” and “max\_lat” are not *-999.0*, “min\_lat” must be smaller than “max\_lat”.
* center\_lon [REAL, DIMENSION|(:) or (:,:)|; IN]: The longitude (X) value of the center of each grid cell. If “dim\_size2” is larger than 3, the array size of “center\_lon” can be “dim\_size1” or “dim\_size1”\*“dim\_size2” (the grid size); otherwise, the array size of “center\_lon” must be “dim\_size1”.
* center\_lat [REAL, DIMENSION|(:) or (:,:)|; IN]: The latitude (Y) value of the center of each grid cell. If “dim\_size2” is larger than 3 and the array size of “center\_lon” is “dim\_size1”, the array size of “center\_lat” must be “dim\_size2”; otherwise, “center\_lat” must have the same array size with “center\_lon”. “center\_lat” must have the same number of dimensions with “center\_lon”.
* mask [INTEGER, DIMENSION|(:) or (:,:)|, OPTIONAL; IN]: The mark to specify each grid cell active or not. The “mask” value of *1* means active while value of *0* means inactive. The array size of “mask” must be the grid size. “mask” must have the same number of dimensions with “center\_lon”. When “mask” is not provided, it means all grid cells are active.
* area [REAL, DIMENSION|(:) or (:,:)|, OPTIONAL; IN]: The area of each grid cell. The array size of “area” must be the grid size. “area” must have the same number of dimensions with “center\_lon”. The unit of “area” should be consistent with “center\_lon”.
* vertex\_lon [REAL, DIMENSION|(:,:) or (:,:,:)|, OPTIONAL; IN]: The longitude (X) values of the vertexes of each grid cell. Corresponding to “center\_lon”, “vertex\_lon” must have an additional lowest dimension whose size is the maximum number of vertexes. *-999.0* can be used to specify the missing values.
* vertex\_lat [REAL, DIMENSION|(:,:) or (:,:,:)|, OPTIONAL; IN]: The latitude (Y) values of the vertexes of each grid cell. Corresponding to “center\_lat”, “vertex\_lat” must have an additional lowest dimension whose size is the maximum number of vertexes. *-999.0* can be used to specify the missing values. “vertex\_lon” and “vertex\_lat” must be provide at the same time and have the same size at the lowest dimension.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

All floating-point (REAL) parameters must have the same detailed data type, i.e., single-precision floating point and double-precision floating point.

* **Description of this API**

This API registers a new horizontal (H2D) grid of the component model corresponding to “comp\_id” using the global grid data provided by the component model and returns the ID of the horizontal grid when the registration succeeds. It targets to support any kind of horizontal grids such as unstructured grids and longitude-latitude grids. For an unstructured grid, users are advised to specify the grid size through “dim\_size1” while setting “dim\_size2” to 0, and make the array size of “center\_lon” and “center\_lat” the same as the grid size. For a longitude-latitude grid, users can treat it as an unstructured grid, and can also specify the size of longitude dimension through “dim\_size1” and specify the size of latitude dimension through “dim\_size2” (in such a case, the array size of “center\_lon” and “center\_lat” can be the same as the grid size, or the same as “dim\_size1” and “dim\_size2” respectively). “Mask”, “area”, “vertex\_lon” and “vertex\_lat” are optional parameters. “vertex\_lon” and “vertex\_lat” are required to be provided when this horizontal grid is involved in conservative interpolations in model coupling.

All processes of the component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

## CCPL\_register\_H2D\_grid\_via\_local\_data

* **integer FUNCTION CCPL\_register\_H2D\_grid\_via\_local\_data(comp\_id, grid\_name, edge\_type, coord\_unit, cyclic\_or\_acyclic, grid\_size, num\_local\_cells, local\_cells\_global\_index, min\_lon, max\_lon, min\_lat, max\_lat, center\_lon, center\_lat, mask, area, vertex\_lon, vertex\_lat, decomp\_name, decomp\_id, annotation)**
* return value [INTEGER; OUT]: The ID of the new horizontal (H2D) grid
* comp\_id [INTEGER; IN]: The ID of the given component model that the new grid is associated with.
* grid\_name [CHARACTER; IN]: The name of the new grid. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘\_’.
* edge\_type [CHARACTER; IN]: The type of the edges of grid cells (“LON\_LAT”， “XY”, “GREAT\_ARC”, or “TriPolar”).
* coord\_unit [CHARACTER; IN]: The unit of the coordinates of the new grid. When the edge type is “LON\_LAT”, “GREAT\_ARC” or “TriPolar”, the unit must be “degrees” or “radians”.
* cyclic\_or\_acyclic [CHARACTER; IN]: whether the longitude (X) dimension is cyclic or acyclic. The value must be “cyclic” or “acyclic”.
* grid\_size [INTEGER; IN]: The size of the size of the global grid (must be larger than 3).
* num\_local\_cells [INTEGER; IN]: The number of grid cells (≥0) corresponding to the local data of the current process.
* local\_cells\_global\_index [INTEGER, DIMENSION(:); IN]: The global index of the grid cells corresponding to the local data of the current process. The array size of “local\_cells\_global\_index” cannot be smaller than “num\_local\_cells”. Each value in “local\_cells\_global\_index” must be between *1* and the “grid\_size”.
* min\_lon [REAL; IN]: Minimum longitude value of the boundary of the domain of the global grid. Its value can be a specific value between *-360.0* and *360.0* degrees (or between *-2PI* and *2PI*) or be *-999.0*. The value of *-999.0* indicates that the component model wants C-Coupler2 to calculate “min\_lon” automatically. Please note that, “min\_lon” calculated by C-Coupler2 may be incorrect.
* max\_lon [REAL; IN]: Maximum longitude value of the boundary of the domain of the global grid. Its value can be a specific value between *-360.0* and *360.0* degrees (or between *-2PI* and *2PI*) or be *-999.0*. The value of *-999.0* indicates that the component model wants C-Coupler2 to calculate “max\_lon” automatically. Please note that, “max\_lon” calculated by C-Coupler2 may be incorrect. “min\_lon” and “max\_lon” must be *-999.0* or not *-999.0* at the same time. When both “min\_lon” and “max\_lon” are not *-999.0*, “min\_lon” can be larger or smaller than “max\_lon”.
* min\_lat [REAL; IN]: Minimum latitude value of the boundary of the domain of the global grid. Its value can be a specific value between *-90.0* and *90.0* degrees (or between *–PI/2* and *PI/2*) or be *-999.0*. The value of *-999.0* indicates that the component model wants C-Coupler2 to calculate “min\_lat” automatically. Please note that, “min\_lat” calculated by C-Coupler2 may be incorrect. When “min\_lat” has been set to *-90* degrees (or *–PI/2*), it indicates that the global grid covers the South Pole.
* max\_lat [REAL; IN]: Maximum latitude value of the boundary of the domain of the global grid. Its value can be a specific value between *-90.0* and *90.0* degrees (or between *–PI/2* and *PI/2*) or be *-999.0*. The value of *-999.0* indicates that the component model wants C-Coupler2 to calculate “max\_lat” automatically. Please note that, “max\_lat” calculated by C-Coupler2 may be incorrect. When “max\_lat” has been set to *90* degrees (or *PI/2*), it indicates that the global grid covers the North Pole. When both “min\_lat” and “max\_lat” are not *-999.0*, “min\_lat” must be smaller than “max\_lat”.
* center\_lon [REAL, DIMENSION(:); IN]: The longitude (X) value of the center of each grid cell. The array size of “center\_lon” must be the same with “num\_local\_cells”.
* center\_lat [REAL, DIMENSION(:); IN]: The latitude (Y) value of the center of each grid cell. The array size of “center\_lat” must be the same with “num\_local\_cells”.
* mask [INTEGER, DIMENSION(:), OPTIONAL; IN]: The mark to specify each local grid cell active or not. The “mask” value of 1 means active while value of 0 means inactive. The array size of “mask” must be the same with “num\_local\_cells”. When “mask” is not provided, it means all local grid cells are active.
* area [REAL, DIMENSION(:), OPTIONAL; IN]: The area of each local grid cell. The array size of “area” must be the same with “num\_local\_cells”. The unit of “area” should be consistent with “center\_lon”.
* vertex\_lon [REAL, DIMENSION(:,:), OPTIONAL; IN]: The longitude (X) values of the vertexes of each local grid cell. Corresponding to “center\_lon”, “vertex\_lon” must have an additional lowest dimension whose size is the maximum number of vertexes. “-999.0” can be used to specify the missing values.
* vertex\_lat [REAL, DIMENSION(:,:), OPTIONAL; IN]: The latitude (Y) values of the vertexes of each grid cell. Corresponding to “center\_lat”, “vertex\_lat” must have an additional lowest dimension whose size is the maximum number of vertexes. “-999.0” can be used to specify the missing values. “vertex\_lon” and “vertex\_lat” must be provide at the same time and have the same size of the lowest dimension.
* decomp\_name [CHARACTER, OPTIONAL; IN]: The name of the new parallel decomposition to be registered at the same time. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘\_’.
* decomp\_id [CHARACTER, OPTIONAL; OUT]: used to return the ID of the new parallel decomposition to be registered at the same time. “decomp\_id” and “decomp\_name” must be provided or not provided at the same time.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

All floating-point (REAL) parameters must have the same detailed data type, i.e., single-precision floating point and double-precision floating point.

* **Description of this API**

This API registers a new horizontal (H2D) grid of the component model corresponding to “comp\_id” using the local grid data provided by the component model and returns the ID of the horizontal grid when the registration succeeds. It is similar to the API CCPL\_register\_H2D\_grid\_via\_global\_data, while using the local grid data of the current process for grid registration. The local grid data are described based on the parameters “num\_local\_cells” and “local\_cells\_global\_index”. This API can also register the corresponding parallel decomposition on the new horizontal grid at the same time.

All processes of the component model are required to call this API at the same time, with consistent parameters (the global information including “grid\_name”, “edge\_type”, “coord\_unit”, “cyclic\_or\_acyclic”, “grid\_size”, “min\_lon”, “max\_lon”, “min\_lat” and “max\_lat” must be the same across all processes). This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

## CCPL\_register\_H2D\_grid\_via\_file

* **Integer FUNCTION CCPL\_register\_H2D\_grid\_via\_file(comp\_id, grid\_name, data\_file\_name, annotation)**
* return value [INTEGER; OUT]: The ID of the new horizontal (H2D) grid.
* comp\_id [INTEGER; IN]: The ID of the given component model that the new grid is associated with.
* grid\_name [CHARACTER; IN]: The name of the new grid. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘\_’.
* data\_file\_name [CHARACTER; IN]: The name of the file (only NetCDF file currently) with grid data. The corresponding file should be included under the directory XXX, and “data\_file\_name” should not include directory. “data\_file\_name” has a maximum length of 1000 characters. Please refer to the examples section for the detailed requirements for the grid data file.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API registers a new horizontal (H2D) grid of the component model corresponding to “comp\_id” based on the grid data in a file and returns the ID of the horizontal grid when the registration succeeds. This API first reads the grid data available in the file and next calls the function CCPL\_register\_H2D\_grid\_via\_global\_data to further register the new horizontal grid. Therefore the grid data in the file must satisfy the corresponding rules of CCPL\_register\_H2D\_grid\_via\_global\_data except that there is no restriction of dimensions of the variables in the grid data file.

All processes of the component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

* **Examples**

Here we give 3 examples with the header information of the NetCDF files.

1. **Example 1**

Example1 is about a longitude-latitude grid. Dimension “lon” in the grid data file corresponds to parameter “dim1\_size” of CCPL\_register\_H2D\_grid\_via\_global\_data; dimension “lon” corresponds to parameter “dim2\_size”; variable “lon” corresponds to parameter “center\_lon”; variable “lat” corresponds to parameter “center\_lat”; the unit of “lon”/“lat”/ “vertex\_lon”/“vertex\_lat” corresponds to parameter “unit”; each of remaining variables and global attributes corresponding to the parameter of CCPL\_register\_H2D\_grid\_via\_global\_data with the same name.

1. **Example 2**

Example2 is similar to example1, with the difference that exmaple2 enumerates the coordinate values of the each grid cells.

1. **Example 3**

Example3 is similar to example1, with the difference that dimensions “lon” and “lat” are replaced by “grid\_size” in exmaple3. Dimension “grid\_size” in the grid data file corresponds to parameter “dim1\_size” of CCPL\_register\_H2D\_grid\_via\_global\_data, while parameter “dim2\_size” should be 0.

dimensions:

lon = 128 ;

lat = 60 ;

num\_vertexes\_lon = 2 ;

num\_vertexes\_lat = 2 ;

variables:

float lon(lon) ;

lon:long\_name = "longitude" ;

lon:unit = "degrees" ;

float vertex\_lon(lon, num\_vertexes\_lon) ;

vertex\_lon:long\_name = "longitude" ;

vertex\_lon:unit = "degrees" ;

float lat(lat) ;

lat:long\_name = "latitude" ;

lat:unit = "degrees" ;

float vertex\_lat(lat, num\_vertexes\_lat) ;

vertex\_lat:long\_name = "latitude" ;

vertex\_lat:unit = "degrees" ;

int mask(lat, lon) ;

float area(lat, lon) ;

// global attributes:

:edge\_type = "LON\_LAT" ;

:cyclic\_or\_acyclic = "cyclic" ;

:min\_lon = 0. ;

:max\_lon = 360. ;

:min\_lat = -90. ;

:max\_lat = 90. ;

**Example1**

dimensions:

lon = 128 ;

lat = 60 ;

num\_vertexes\_lon = 4 ;

num\_vertexes\_lat = 4 ;

variables:

float lon(lat, lon) ;

lon:long\_name = "longitude" ;

lon:unit = "degrees" ;

float vertex\_lon(lat, lon, num\_vertexes\_lon) ;

vertex\_lon:long\_name = "longitude" ;

vertex\_lon:unit = "degrees" ;

float lat(lat, lon) ;

lat:long\_name = "latitude" ;

lat:unit = "degrees" ;

float vertex\_lat(lat, lon, num\_vertexes\_lat) ;

vertex\_lat:long\_name = "latitude" ;

vertex\_lat:unit = "degrees" ;

int mask(lat, lon) ;

float area(lat, lon) ;

// global attributes:

:edge\_type = "LON\_LAT" ;

:cyclic\_or\_acyclic = "cyclic" ;

:min\_lon = 0. ;

:max\_lon = 360. ;

:min\_lat = -90. ;

:max\_lat = 90. ;

**Example2**

## 

dimensions:

grid\_size = 7680 ;

num\_vertexes\_lon = 4 ;

num\_vertexes\_lat = 4 ;

variables:

float lon(grid\_size) ;

lon:long\_name = "longitude" ;

lon:unit = "degrees" ;

float vertex\_lon(grid\_size, num\_vertexes\_lon) ;

vertex\_lon:long\_name = "longitude" ;

vertex\_lon:unit = "degrees" ;

float lat(grid\_size) ;

lat:long\_name = "latitude" ;

lat:unit = "degrees" ;

float vertex\_lat(grid\_size, num\_vertexes\_lat) ;

vertex\_lat:long\_name = "latitude" ;

vertex\_lat:unit = "degrees" ;

int mask(grid\_size) ;

float area(grid\_size) ;

// global attributes:

:edge\_type = "LON\_LAT" ;

:cyclic\_or\_acyclic = "cyclic" ;

:min\_lon = 0. ;

:max\_lon = 360. ;

:min\_lat = -90. ;

:max\_lat = 90. ;

**Example3**

## Function CCPL\_register\_H2D\_grid\_from\_another\_component

* **Integer FUNCTION CCPL\_register\_H2D\_grid\_from\_another\_component(comp\_id, grid\_name, annotation)**
* return value [INTEGER; OUT]: The ID of the new horizontal (H2D) grid.
* comp\_id [INTEGER; IN]: The ID of the given component model that the new grid is associated with.
* grid\_name [CHARACTER; IN]: The name of the new grid. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘\_’.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API registers a new horizontal (H2D) grid of the component model corresponding to “comp\_id” through copying an H2D grid that has been successfully registered in another component model. The corresponding H2D grid in another component model is specified in the *redirection configuration file* of the component model corresponding to “comp\_id” (please refer to section XXX). Please refer to the examples section for the specification of an H2D grid in another component model.

All processes of the component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

* **Examples**

The specification of an H2D grid in another component model corresponding to one entry in section “*component\_grid\_redirection\_configuration*” of the *redirection configuration file*. “local\_grid\_name” means the name of the new H2D grid to be registered. “another\_comp\_full\_name” means the full name of the another component model and “another\_comp\_grid\_name” means the name of the corresponding H2D grid in the another component model.

<component\_grid\_redirection\_configuration>

<entry local\_grid\_name="remote\_atm\_grid" another\_comp\_full\_name="gamil" another\_comp\_grid\_name="gamil\_H2D\_grid" />

</component\_grid\_redirection\_configuration>

## CCPL\_register\_V1D\_Z\_grid\_via\_model\_data

* **Integer FUNCTION CCPL\_register\_V1D\_Z\_grid\_via\_model\_data(comp\_id, grid\_name, coord\_unit, coord\_values, annotation)**
* return value [INTEGER; OUT]: The ID of the new vertical (V1D) grid.
* comp\_id [INTEGER; IN]: The ID of the given component model that the new grid is associated with.
* grid\_name [CHARACTER; IN]: The name of the new grid. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘\_’.
* coord\_unit [CHARACTER; IN]: The unit of the coordinates of the new grid. It has a maximum length of 80 characters.
* coord\_values [REAL, DIMENSION(:); IN]: The coordinate values of the vertical Z grid. The size of the vertical grid is determined by the array size of “coord\_values”. The values in “coord\_values” must be in an ascending or descending order.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API registers a vertical (V1D) Z grid of the component model corresponding to “comp\_id” using the global grid data provided by the model and returns the ID of the vertical Z grid when the registration succeeds. All processes of the component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

## CCPL\_register\_V1D\_SIGMA\_grid\_via\_model\_data

* **Integer FUNCTION CCPL\_register\_V1D\_SIGMA\_grid\_via\_model\_data(comp\_id, grid\_name, coord\_unit, top\_value, sigma\_values, annotation)**
* return value [INTEGER; OUT]: The ID of the new vertical (V1D) grid.
* comp\_id [INTEGER; IN]: The ID of the given component model that the new grid is associated with.
* grid\_name [CHARACTER; IN]: The name of the new grid. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘\_’.
* coord\_unit [CHARACTER; IN]: The unit of the coordinates of the new grid. It has a maximum length of 80 characters.
* top\_value [REAL; IN]: The value corresponding to the top level of the SIGMA grid
* sigma\_values [REAL, DIMENSION(:); IN]: The SIGMA values of the vertical SIGMA grid. The size of the vertical grid is determined by the array size of “sigma\_values”. The values in “sigma\_values” must be in an ascending or descending order.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

All floating-point (REAL) parameters must have the same detailed data type, i.e., single-precision floating point and double-precision floating point.

* **Description of this API**

This API registers a vertical (V1D) SIGMA grid of the component model corresponding to “comp\_id” using the global grid data provided by the model and returns the ID of the vertical SIGMA grid when the registration succeeds. All processes of the component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

## CCPL\_register\_V1D\_HYBRID\_grid\_via\_model\_data

* **Integer FUNCTION CCPL\_register\_V1D\_HYBRID\_grid\_via\_model\_data(comp\_id, grid\_name, coord\_unit, top\_value, coef\_A, coef\_B, annotation)**
* return value [INTEGER; OUT]: The ID of the new vertical (V1D) grid.
* comp\_id [INTEGER; IN]: The ID of the given component model that the new grid is associated with.
* grid\_name [CHARACTER; IN]: The name of the new grid. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘\_’.
* coord\_unit [CHARACTER; IN]: The unit of the coordinates of the new grid. It has a maximum length of 80 characters.
* top\_value [REAL; IN]: The value corresponding to the top level of the HYBRID grid.
* coef\_A [REAL, DIMENSION(:); IN]: The values of coefficients A of the vertical HYBRID grid. The size of the vertical grid is determined by the array size of “coef\_A”. The values in “coef\_A” must be in an ascending or descending order.
* coef\_B [REAL, DIMENSION(:); IN]: The values of coefficients B of the vertical HYBRID grid. “coef\_B” must have the same array size with “coef\_A”. The values in “coef\_B” must be in an ascending or descending order.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

All floating-point (REAL) parameters must have the same detailed data type, i.e., single-precision floating point and double-precision floating point.

* **Description of this API**

This API registers a vertical (V1D) HYBRID grid of the component model corresponding to “comp\_id” using the global grid data provided by the model and returns the ID of the vertical HYBRID grid when the registration succeeds. All processes of the component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

## CCPL\_register\_MD\_grid\_via\_multi\_grids

* **Integer FUNCTION CCPL\_register\_MD\_grid\_via\_multi\_grids(comp\_id, grid\_name, sub\_grid1\_id, sub\_grid2\_id, sub\_grid3\_id, mask, annotation)**
* return value [INTEGER; OUT]: The ID of the new multi-dimension (MD) grid.
* comp\_id [INTEGER; IN]: The ID of the given component model that the new grid is associated with.
* grid\_name [CHARACTER; IN]: The name of the new grid. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘\_’.
* sub\_grid1\_id [INTEGER; IN]: The ID of the first sub grid that has already been registered. The first sub grid must be associated with the given component model.
* sub\_grid2\_id [INTEGER; IN]: The ID of the second sub grid that has already been registered. The second sub grid must be also associated with the given component model, and do not have common coordinates with the first sub grid.
* sub\_grid3\_id [INTEGER, OPTIONAL; IN]: The ID of the third sub grid that has already been registered. The third sub grid must be also associated with the given component model, and do not have common coordinates with the first and second sub grids.
* mask [INTEGER, DIMENSION(:), OPTIONAL; IN]: The mark to specify each grid cell active or not. The “mask” value of 1 means active while value of 0 means inactive. The array size of “mask” must be the size of the multiple-dimension grid. When “mask” is not provided, it means all grid cells are active.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API registers a new multi-dimension (MD) grid using several grids that have already been registered and returns the ID of the new grid when the registration succeeds. The new grid corresponds to the same component model with all sub grids. All processes of the given component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

## CCPL\_register\_mid\_point\_grid

* **SUBROUTINE CCPL\_register\_mid\_point\_grid (level\_3D\_grid\_id, mid\_3D\_grid\_id, mid\_V1D\_grid\_id, mask, annotation)**
* level\_3D\_grid\_id [INTEGER; IN]: The ID of the interface-level grid that has already been registered to C-Coupler. The interface-level grid is a 3-D grid consisting of a horizontal grid and a vertical grid. The interface-level grid cannot be the middle-point grid of another grid.
* mid\_3D\_grid\_id [INTEGER; OUT]: Return the ID of the middle-point grid.
* mid\_V1D\_grid\_id [INTEGER; OUT]: Return the ID of the vertical sub grid of the middle-point grid.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API registers the middle-point grid of an interface-level grid (3-D) and returns the ID of the middle-point rid and the ID of the vertical sub grid of the middle-point grid. These two newly registered grids are associated to the same component model with the interface-level grid. All processes of the component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

## CCPL\_set\_3D\_grid\_dynamic\_surface\_field

* **SUBROUTINE CCPL\_set\_3D\_grid\_dynamic\_surface\_field(grid\_id, field\_id, annotation)**
* grid\_id [INTEGER; IN]: The ID of the 3-D grid that has already been registered to C-Coupler. The 3-D grid consists of a horizontal grid and a vertical grid that is a SIGMA grid or HYBRID grid.
* field\_id [INTEGER; IN]: The ID of the surface field instance that has already been registered to C-Coupler. This field instance must be on the horizontal sub grid of the 3-D grid, and corresponds to the same component model with the 3-D grid.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API registers the dynamic surface field for a 3-D grid. Dynamic surface field means that its values change with the time integration, like the surface pressure in an atmospheric model. All processes of the corresponding component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

## CCPL\_set\_3D\_grid\_static\_surface\_field

* **SUBROUTINE CCPL\_set\_3D\_grid\_static\_surface\_field(grid\_id, field\_id, annotation)**
* grid\_id [INTEGER; IN]: The ID of the 3-D grid that has already been registered to C-Coupler. The 3-D grid consists of a horizontal grid and a vertical grid that is a SIGMA grid or HYBRID grid.
* field\_id [INTEGER; IN]: The ID of the surface field instance that has already been registered to C-Coupler. This field instance must be on the horizontal sub grid of the 3-D grid, and corresponds to the same component model with the 3-D grid.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API registers the static surface field for a 3-D grid. Static surface field means that its values keeps constant with the time integration, like the depth of the bottom of oceans in an ocean model. All processes of the corresponding component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

## CCPL\_set\_3D\_grid\_external\_surface\_field

* **SUBROUTINE CCPL\_set\_3D\_grid\_external\_surface\_field(grid\_id, annotation)**
* grid\_id [INTEGER; IN]: The ID of the 3-D grid that has already been registered to C-Coupler. The 3-D grid consists of a horizontal grid and a vertical grid that is a SIGMA grid or HYBRID grid.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API declares that the surface field of a 3-D grid is external. External surface field means that its values are determined by the source grid in data interpolation (a 3-D grid with external surface field cannot be used as a source grid in data interpolation). All processes of the corresponding component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

## CCPL\_get\_grid\_size

* **integer FUNCTION CCPL\_get\_grid\_size(grid\_id, annotation)**
* return value [INTEGER; OUT]: The size of the grid.
* grid\_id [INTEGER; IN]: The ID of the grid that has already been registered to C-Coupler.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API returns the size of a grid that has already been registered to C-Coupler.

## CCPL\_get\_grid\_id

* **integer FUNCTION CCPL\_get\_grid\_id(comp\_id, grid\_name, annotation)**
* return value [INTEGER; OUT]: The ID of the new grid.
* comp\_id [INTEGER; IN]: The ID of the given component model.
* grid\_name [CHARACTER; IN]: The name of the grid. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘\_’.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API will return the ID of the corresponding grid when it has already been registered to C-Coupler or return -1 when the corresponding grid has not been registered. It does not require all processes to call this API.

## CCPL\_get\_H2D\_grid\_data

* **SUBROUTINE CCPL\_get\_H2D\_grid\_data (grid\_id, decomp\_id, label, grid\_data, annotation)**
* grid\_id [INTEGER; IN]: The ID of the given horizontal (H2D) grid that has already been registered to C-Coupler.
* decomp\_id [INTEGER; IN]: The ID of a given parallel decomposition on the horizontal grid. When local grid data is wanted, a valid “decomp\_id” should be given, and the corresponding parallel decomposition must be associated with the given horizontal grid; when global grid data is wanted, “decomp\_id” should be set to -1.
* label [CHARACTE; IN]: The label of the grid data. The valid labels currently supported include “lon”, “lat”, and “mask”. “lon” means the longitude (X) values of all grid cells. “lat” means the latitude (Y) values of all grid cells. “mask” means the mask values of all grid cells.
* grid\_data [REAL|INTEGER, DIMENSION(:); out]: Return the grid data. The array size of “grid\_data” must be consistent with given parallel decomposition. The data type of “grid\_data” must be consistent with “label” (for example, when “label” is “mask”, “grid\_data” must be an integer array).
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API gets the grid data of a horizontal (H2D) grid that has already been registered to C-Coupler2.

## API for parallel decomposition management

To make a model accelerated by a modern high-performance computer that generally includes a large number of processor cores, the model needs to be parallelized with MPI, where the domains of model grids are decomposed into separate subdomains for parallel integration. To make model coupling handled in parallel, C-Coupler2 also manages such kind of decomposition (called parallel decomposition) and provides APIs to enable active component models to register their parallel decompositions to C-Coupler2.

Currently, C-Coupler2 only supports the parallel decompositions on horizontal grids, which means that the parallel decomposition further on vertical grids are not supported yet. Therefore, a parallel decomposition is associated with a horizontal grid, so as to be associated with the component model corresponding to the horizontal grid. The keyword for a parallel decomposition can be expressed as <ID of component model, decomposition name>. Therefore, different parallel decompositions in the same component model cannot have the same name, while the parallel decompositions in different component models can have the same name. There could be multiple parallel decompositions on the same horizontal grid.

In C-Coupler2, a parallel decomposition on a horizontal grid is described through enumerating global grid cell indexes of the local grid cells assigned to each process of the corresponding component model. A valid global grid cell index should be between 1 and the size of the horizontal grid. For the local grid cells that are unnecessary to be considered in model coupling (for example, the land only grid cells in an ocean model), the corresponding values of global grid cell index can be set to a C-Coupler2 pre-defined variable *CCPL\_NULL\_INT*, to save some overhead in model coupling.

Table X lists out the APIs for parallel decomposition management as well as their brief descriptions.

Table X APIs for parallel decomposition management.

|  |  |  |
| --- | --- | --- |
| No. | API | Brief description |
| 1 | CCPL\_register\_parallel\_decomp | Register a parallel decomposition on a horizontal grid |

## CCPL\_register\_parallel\_decomp

* **integer FUNCTION CCPL\_register\_parallel\_decomp(decomp\_name, grid\_id, num\_local\_cells, local\_cells\_global\_index, annotation)**
* return value [INTEGER; OUT]: The ID of the parallel decomposition.
* decomp\_name [CHARACTER; IN]: The name of the parallel decomposition. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘\_’.
* grid\_id [INTEGER; IN]: The ID of the corresponding horizontal (H2D) grid that has already been registered to C-Coupler.
* num\_local\_cells [INTEGER; IN]: The number of local grid cells (≥0) in the parallel decomposition of the current process.
* local\_cells\_global\_index [INTEGER, DIMENSION(:); IN]: The global index of the local grid cells in the parallel decomposition of the current process. The array size of “local\_cells\_global\_index” cannot be smaller than “num\_local\_cells”.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API registers a new parallel decomposition of an H2D grid (“grid\_id”) among all processes of the component model corresponding to “grid\_id”, and returns the ID of the new parallel decomposition when the registration succeeds. The new parallel decomposition corresponds to the same component model with “grid\_id”. All processes of the component model are required to call this API at the same time, with the same “decomp\_name” and the same horizontal grid. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

## API for coupling field management

C-Coupler2 employs a concept of coupling field instance to manage coupling fields. A coupling field instance includes a set of META information and a memory buffer that keeps the data values of an instance of a coupling field. A field instance is associated with a unique component model, a unique grid and a unique parallel decomposition. Moreover, an attribute of “buf\_mark” is employed in each field instance, to separate the multiple field instances that are in the same component model, on the same grid and on the same parallel decomposition (for example, as land surface, oceans and sea ice are under the bottom of atmosphere, an atmosphere model may receive multiple coupling field instances of surface temperature each of which is from a land surface model, an ocean model and a sea ice model respectively). Therefore, the keyword for a field instance can be expressed as <ID of component model, ID of grid, ID of parallel decomposition, buf\_mark>. For a scalar field instance that is not on a grid, the corresponding grid ID and parallel decomposition ID should be set to *-1*.

Based on C-Coupler2 APIs, a component model can register field instances to C-Coupler2, in order for providing, obtaining and remapping coupling fields in model coupling. Moreover, a component model can also register field instances for I/O. In the current implementation of C-Coupler2, a component model should constantly keep the memory buffers (for example, arrays) that have been registered as field instances until C-Coupler2 has been finalized.

Table X lists out the APIs for coupling field management as well as their brief descriptions.

Table X APIs for coupling field management.

|  |  |  |
| --- | --- | --- |
| No. | API | Brief description |
| 1 | CCPL\_register\_field\_instance | Register a coupling field instance |
| 2 | CCPL\_register\_IO\_field\_from\_field\_instance | Register an I/O field from a coupling field instance |
| 3 | CCPL\_register\_IO\_fields\_from\_field\_instances | Register a set of I/O fields from a set of coupling field instances |
| 4 | CCPL\_register\_IO\_field\_from\_data\_buffer | Register an I/O field from a data buffer of a given component model |

## CCPL\_register\_field\_instance

* **integer FUNCTION CCPL\_register\_field\_instance(data\_buf, field\_name, decomp\_id, comp\_or\_grid\_id, buf\_mark, field\_unit, annotation)**
* Return value [INTEGER; OUT]: The ID of the new coupling field instance.
* data\_buf [REAL or INTEGER, no DIMENSION or DIMENSION|(:), (:,:), (:,:,:) or (:,:,:,:)|; INOUT]: The model data buffer corresponding to the coupling field instance.
* field\_name [CHARACTER; IN]: The name of the coupling field. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘\_’. Different field instances can share the same “field\_name”. A “field\_name” is legal only when there is a corresponding entry in the configuration file “public\_field\_attribute.xml” (Please refer to Section XXX for details).
* decomp\_id [INTEGER; IN]: If the field instance is a scalar variable or the field instance is not on a grid with horizontal sub grid (for example, the field instance is only on a vertical grid), “decomp\_id” is -*1*; otherwise, “decomp\_id” is the ID of the corresponding parallel decomposition that has already been registered to C-Coupler2.
* comp\_or\_grid\_id [INTEGER; IN]: If the field instance is a scalar variable, “comp\_or\_grid\_id” is the ID of the corresponding component model; otherwise, “comp\_or\_grid\_id” is the ID of the corresponding grid that has already been registered to C-Coupler2. When “comp\_or\_grid\_id” is the ID of a grid, “decomp\_id” and “comp\_or\_grid\_id” must correspond to the same component model, and the horizontal grid corresponding to “decomp\_id” must be a sub grid of the grid corresponding to “comp\_or\_grid\_id”.
* buf\_mark [INTEGER; IN]: a mark used to separate different coupling field instances with the same field name, the same “decomp\_id” and the same “comp\_or\_grid\_id”. “buf\_mark” must be a non-negative integer.
* field\_unit [CHARACTER, OPTIONAL; IN]: The unit of the coupling field instance. Default unit specified in the configuration file “public\_field\_attribute.xml” Please refer to Section XXX for details) will be used when “field\_unit” is not specified when calling this API.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API uses a model data buffer to register a coupling field instance, and returns the ID of the new coupling field instance when the registration succeeds. “field\_name”, “decomp\_id”, “comp\_or\_grid\_id” and “buf\_mark” are keywords of a coupling field instance, which means two coupling field instances cannot share the same “field\_name”, “decomp\_id”, “comp\_or\_grid\_id” and “buf\_mark”. The new parallel decomposition corresponds to the same component model with “comp\_or\_grid\_id”. All processes of the component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

## CCPL\_register\_IO\_field\_from\_field\_instance

* **SUBROUTINE CCPL\_register\_IO\_field\_from\_field\_instance(field\_inst\_id, field\_IO\_name, annotation)**
* field\_inst\_id [INTEGER; IN]: The ID of the corresponding field instance that has already been registered to C-Coupler.
* field\_IO\_name [CHARACTER, OPTIONAL; IN]: The variable name when writing the corresponding field instance into a NETCDF file. When “field\_IO\_name” is not specified, the variable name is the “field\_name” of the corresponding field instance.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API registers an I/O field based on a field instance that has already been registered to C-Coupler. The variable name in the NETCDF file is treated as the keyword of an I/O field. All processes of the component model corresponding to “field\_inst\_id” are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

## CCPL\_register\_IO\_fields\_from\_field\_instances

* **SUBROUTINE CCPL\_register\_IO\_fields\_from\_field\_instances(num\_field\_inst, field\_inst\_ids, annotation)**
* num\_field\_inst [INTEGER; IN]: The number of field instances that will be used to register I/O fields.
* field\_inst\_ids [INTEGER, DIMENSION(:); IN]: The IDs of the field instances that have already been registered to C-Coupler. All fields must correspond to the same component model.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.

This API registers a number of I/O fields based on the corresponding field instances that have already been registered to C-Coupler. For a field instance, its field name is used as the variable name in the NETCDF file, which is treated as the keyword of the I/O field. All processes of the component model corresponding to “field\_inst\_ids” are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

## SUBROUTINE CCPL\_register\_IO\_field\_from\_data\_buffer

* **SUBROUTINE CCPL\_register\_IO\_field\_from\_data\_buffer(data\_buf, field\_IO\_name, field\_long\_name, field\_unit, comp\_or\_grid\_id, decomp\_id, annotation)**
* data\_buf [REAL or INTEGER, no DIMENSION or DIMENSION|(:), (:,:), (:,:,:) or (:,:,:,:)|; INOUT]: The model data buffer corresponding to the I/O field.
* field\_IO\_name [CHARACTER; IN]: The variable name when writing the I/O field into a NETCDF file.
* field\_long\_name [CHARACTER; IN]: The long name of the variable when writing the I/O field into a NETCDF file.
* field\_unit [CHARACTER; IN]: The unit of the variable when writing the I/O field into a NETCDF file.
* decomp\_id [INTEGER; IN]: if the field instance is a scalar variable, “decomp\_id” is -1; otherwise, “decomp\_id” is the ID of the corresponding parallel decomposition that has already been registered to C-Coupler.
* comp\_or\_grid\_id [INTEGER; IN]: if the field instance is a scalar variable, “comp\_or\_grid\_id” is the ID of the corresponding component; otherwise, “comp\_or\_grid\_id” is the ID of the corresponding grid that has already been registered to C-Coupler. When “comp\_or\_grid\_id” is the ID of a grid, “decomp\_id” and “comp\_or\_grid\_id” must correspond to the same component, and the horizontal grid corresponding to “decomp\_id” must be the same with or a sub grid of the grid corresponding to “comp\_or\_grid\_id”.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API registers an I/O field based on a data buffer of the corresponding component model. The variable name in the NETCDF file is treated as the keyword of an I/O field. All processes of the component model corresponding to “field\_inst\_id” are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

## API for coupling interface management

In C-Coupler2, the procedures for coupling are triggered by coupling interfaces. A coupling interface is associated to a unique active component model. The keyword for a coupling interface can be expressed as <ID of component model, interface name>. Therefore, different coupling interfaces in the same component model cannot have the same interface name, while the coupling interfaces in different component models can have the same interface name.

Coupling interfaces are classified into three types: import interfaces, export interfaces and remap interfaces. An import interface enables a component model to obtain coupling fields from itself or other component models in the coupled system. An export interface enables a component model to provide a number of coupling fields to the coupled system. A remap interface enables a component model to remap its coupling fields from a source grid to a target grid. There are two detailed kinds of remap interfaces: normal remap interface and fraction based remap interface. A normal remap interface directly interpolates coupling fields from the source grid to target grid, while a fraction based remap interface will first adjust the values of coupling fields on the source grid based on the source fraction before remapping and finally adjust the values of coupling fields on the target grid based on the target fraction after remapping (the source fraction is also remapped from the source grid to the target grid, in order to produce the target fraction at the same time). Fraction based remap interfaces are generally necessary for guaranteeing conservation in model coupling.

There are three steps to utilize a coupling interface. The first step is to register the coupling interface, where a timer is required to be specified for controlling the execution of the coupling interface in time integration. The second step is to generate coupling procedures for the coupling interface. The third step is to execute the coupling interface. Although the API to execute a coupling interface can be called at each time step, a coupling interface will be truly executed only when its timer is bypassed or its timer is on. C-Coupler2 enables to bypass the timer when executing a coupling interface, in order to achieve flexible coupling at the initialization stage of the coupled system. Please note that the timer of a coupling interface can be bypassed only at the initialization stage (in other words, only at the first initial time step or first restart time step, when the time step has not been advanced), and please note that when the timer of a coupling interface is not bypassed, the coupling interface will be truly executed at most once at the same time step, which means that additional API calls for executing the coupling interface at the same time step will be neglected.

For a remap interface that does not refer to coupling between different coupling interfaces or between different component models, the coupling generator will automatically generate its coupling procedures when registering it. Coupling procedures of an export/import interface are also generated automatically by the coupling generator, but will not be generated when registering the interface, because an export/import interface refers to coupling between different coupling interfaces in the same or different component models. To generate coupling procedures of export/import interfaces, the coupling generator will analyze possible connections from export interfaces to import interfaces based on the name of each coupling field. A coupling connection from an export interface to an import interface can be generated only when these two coupling interfaces have common field names. Regarding a field name, C-Coupler2 allows an export interface to be connected to any number of import interfaces, while forces an import interface be connected from a unique export interface. In other words, each field in an import interface must have only one provider. If there are multiple providers for a field in an import interface, users must specify how to pick out only one provider. C-Coupler2 provides two manners for such specification. The first manner is to declare a fixed connection between an export interface and an import interface when registering these two interfaces. Two coupling interfaces with the fixed connection must have the same set of field names. They will be only connected with each other. In other words, the coupling generator will not connect any of them with any coupling interface beyond them. Their coupling procedures can be generated when the API for connecting the corresponding component models (CCPL\_connect\_fixed\_interfaces) is called, all when the overall coupling generation is invoked by all root component models through calling the API CCPL\_end\_coupling\_configuration. The second manner is to specify coupling connections for an unfixed import interface (without the specification of a fixed coupling connection) through the XML file named as *Comp\_full\_name*.import.redirection.xml (*comp\_full\_name* means the full name of the corresponding component model; please refer to Section 3.1 for details), where each coupling connection corresponds to a distinct subset of field names in the unfixed import interface and different coupling connections can correspond to different unfixed export interfaces in the same or different component models. C-Coupler2 does not require to specify coupling connections for unfixed export interfaces. Their coupling procedures of unfixed interfaces is generated when the overall coupling generation is invoked by the root component models through calling the API CCPL\_end\_coupling\_configuration. Coupling procedures of unfixed import/export interfaces are generated when the overall coupling generation is invoked by all root component models through calling the API CCPL\_end\_coupling\_configuration.

Table X lists out the APIs for parallel decomposition management as well as their brief descriptions.

Table X APIs for coupling interface management.

|  |  |  |
| --- | --- | --- |
| No. | API | Brief description |
| 1 | CCPL\_register\_export\_interface | Register a coupling interface for exporting field instances |
| 2 | CCPL\_register\_import\_interface | Register a coupling interface for importing field instances |
| 3 | CCPL\_register\_normal\_remap\_interface | Registering a coupling interface for remapping field instances normally |
| 4 | CCPL\_register\_frac\_based\_remap\_interface | Registering a coupling interface for remapping field instances where faction will be used |
| 5 | CCPL\_connect\_fixed\_interfaces | Connect fixed coupling interfaces between two given component models |
| 6 | CCPL\_get\_comp\_full\_name\_via\_interface\_tag | Get the full name of a component model corresponding to an interface tag |
| 7 | CCPL\_execute\_interface\_using\_id | Execute a coupling interface specified by an interface ID |
| 8 | CCPL\_execute\_interface\_using\_name | Execute a coupling interface in a component model specified by an interface name |

## CCPL\_register\_export\_interface

* **integer FUNCTION CCPL\_register\_export\_interface(interface\_name, num\_field\_instances, field\_instance\_IDs, timer\_ID, interface\_tag, annotation)**
* Return value [INTEGER; OUT]: The ID of the new export interface.
* interface\_name [CHARACTER; IN]: The name of the new instance. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘\_’.
* num\_field\_instances [INTEGER; IN]: The number of field instances exported via this coupling interface. “num\_field\_instances” must be larger than 0.
* field\_instance\_IDs [INTEGER, DIMENSION(:); IN]: The ID of the field instances that are exported by the same component model. All field instances specified by “field\_instance\_IDs” must correspond to the same component model. The array size of “field\_instance\_IDs” cannot be smaller than “num\_field\_instances”. Any two coupling field instances cannot share the same field name.
* timer\_ID [INTEGER; IN]: The ID of the timer corresponding to the new interface. All source field instances will be exported when the timer is on. “timer\_ID” must correspond to the same component model with all field instances.
* Interface\_tag [CHARACTER, OPTIONAL; IN]: This tag is used to mark that the current export interface is a fixed coupling interface and used to specify the fixed import interface that the current export interface will be connected with. It can be a string formatted as “*comp\_full\_name*$*interface\_name*” where “*interface\_name*” is the name of the fixed import interface and “*comp\_full\_name*” is the full name of the component model corresponding to the fixed import interface. It can also be a simple keyword used to search an entry in the XML file “*comp\_full\_name*.import.redirection.xml”, where “*comp\_full\_name*” is the full name of the component model corresponding to the current export interface (please refer to Section XXX). It has a maximum length of 600 characters.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API registers a new coupling interface that can export a number of field instances produced by the corresponding component model in model coupling, and returns the ID of the new interface when the registration succeeds. All processes of the corresponding component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

## CCPL\_register\_import\_interface

* **integer FUNCTION CCPL\_register\_import\_interface(interface\_name, num\_field\_instances, field\_instance\_IDs, timer\_ID, inst\_or\_aver, interface\_tag, annotation)**
* return value [INTEGER; OUT]: The ID of the new import interface.
* interface\_name [CHARACTER; IN]: The name of the new instance. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘\_’.
* num\_field\_instances [INTEGER; IN]: The number of field instances imported via this coupling interface. “num\_field\_instances” must be larger than 0.
* field\_instance\_IDs [INTEGER, DIMENSION(:); IN]: The ID of the field instances that are imported by the same component model. All field instances specified by “field\_instance\_IDs” must correspond to the same component model. The array size of “field\_instance\_IDs” cannot be smaller than “num\_field\_instances”. Any two coupling field instances cannot share the same field name.
* timer\_ID [INTEGER; IN]: The ID of the timer corresponding to the new interface. All source field instances will be imported when the timer is on. “timer\_ID” must correspond to the same component model with all field instances.
* inst\_or\_aver [INTEGER; IN]: The mark for specifying using average value or instantaneous value when importing the field instances. The value of *1* means using average value, while the value of *0* means using instantaneous value.
* Interface\_tag [CHARACTER, OPTIONAL; IN]: This tag is used to mark that the current export interface is a fixed coupling interface and used to specify the fixed import interface that the current export interface will be connected with. It can be a string formatted as “*comp\_full\_name*$*interface\_name*” where “*interface\_name*” is the name of the fixed import interface and “*comp\_full\_name*” is the full name of the component model corresponding to the fixed import interface. It can also be a simple keyword used to search an entry in the XML file “*comp\_full\_name*.import.redirection.xml”, where “*comp\_full\_name*” is the full name of the component model corresponding to the current export interface (please refer to Section XXX). It has a maximum length of 600 characters.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API registers a new coupling interface that enables a corresponding component model to import a number of field instances from itself or other component models, and returns the ID of the new interface when the registration succeeds. All processes of the corresponding component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

## CCPL\_register\_normal\_remap\_interface

* **integer FUNCTION CCPL\_register\_normal\_remap\_interface(interface\_name, num\_field\_instances, field\_instance\_IDs\_source, field\_instance\_IDs\_target, timer\_ID, inst\_or\_aver, annotation)**
* Return value [INTEGER; OUT]: The ID of the new remapping interface.
* interface\_name [CHARACTER; IN]: The name of the new instance. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘\_’.
* num\_field\_instances [INTEGER; IN]: The number of field instances remapped via this coupling interface. “num\_field\_instances” must be larger than 0.
* field\_instance\_IDs\_source [INTEGER, DIMENSION(:); IN]: The ID of the field instances on the source grids. The array size of “field\_instance\_IDs\_source” cannot be smaller than “num\_field\_instances”. Any two source coupling field instances cannot share the same field name.
* field\_instance\_IDs\_target [INTEGER, DIMENSION(:); IN]: The ID of the field instances on the target grids. The array size of “field\_instance\_IDs\_target” cannot be smaller than “num\_field\_instances”. “field\_instance\_IDs\_target” must be consistent with “field\_instance\_IDs\_source”: the *i*th element in “field\_instance\_IDs\_source” and “field\_instance\_IDs\_target” must correspond to the same field name. All field instances specified by “field\_instance\_IDs\_source” and “field\_instance\_IDs\_target” must correspond to the same component model.
* timer\_ID [INTEGER; IN]: The ID of the timer corresponding to the new interface. All source field instances will remapped to target field instances when the timer is on. “timer\_ID” must correspond to the same component model with all field instances.
* inst\_or\_aver [INTEGER; IN]: The mark for specifying using average value or instantaneous value for remapping. The value of *1* means using average value, while the value of *0* means using instantaneous value.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API registers a new coupling interface for remapping a number of field instances normally and returns the ID of the new interface when the registration succeeds. All processes of the corresponding component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

## CCPL\_register\_frac\_based\_remap\_interface

* **Integer FUNCTION CCPL\_register\_frac\_based\_remap\_interface(interface\_name, num\_field\_instances, field\_instance\_IDs\_source, field\_instance\_IDs\_target, timer\_ID, inst\_or\_aver, frac\_src, frac\_dst, annotation)**
* return value [INTEGER; OUT]: The ID of the new remapping interface.
* interface\_name [CHARACTER; IN]: The name of the new instance. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘\_’.
* num\_field\_instances [INTEGER; IN]: The number of field instances remapped via this coupling interface. “num\_field\_instances” must be larger than 0.
* field\_instance\_IDs\_source [INTEGER, DIMENSION(:); IN]: The ID of the field instances on the source grids. All source field instances must be on the same horizontal grid and on the same parallel decomposition. The array size of “field\_instance\_IDs\_source” cannot be smaller than “num\_field\_instances”. Any two source coupling field instances cannot share the same field name.
* field\_instance\_IDs\_target [INTEGER, DIMENSION(:); IN]: The ID of the field instances on the target grids. All target field instances must be on the same horizontal grid and on the same parallel decomposition. The array size of “field\_instance\_IDs\_target” cannot be smaller than “num\_field\_instances”. “field\_instance\_IDs\_target” must be consistent with “field\_instance\_IDs\_source”: the *i*th element in “field\_instance\_IDs\_source” and “field\_instance\_IDs\_target” must correspond to the same field name. All field instances specified by “field\_instance\_IDs\_source” and “field\_instance\_IDs\_target” must correspond to the same component model.
* timer\_ID [INTEGER; IN]: The ID of the timer corresponding to the new interface. All source field instances will remapped to target field instances when the timer is on. “timer\_ID” must correspond to the same component model with all field instances.
* inst\_or\_aver [INTEGER; IN]: The mark for specifying using average value or instantaneous value for remapping. The value of 1 means using average value, while the value of 0 means using instantaneous value.
* frac\_src [REAL, DIMENSION(:); IN]: The fraction values on the source grid. Its array size must be the same with the field size of each source field instance.
* frac\_dst [REAL, DIMENSION(:), OPTIONAL; OUT]: The fraction values on the target grid. Its array size must be the same with the field size of each target field instance.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API registers a new coupling interface that uses fraction values to remap a number of field instances, and then returns the ID of the new interface when the registration succeeds. The remapping weights for each coupling field must be the same. All processes of the corresponding component model are required to call this API at the same time, with consistent parameters. This API cannot be called when the coupling configuration stage of the corresponding component model has already been ended.

## CCPL\_get\_comp\_full\_name\_via\_interface\_tag

* **logical FUNCTION CCPL\_get\_comp\_full\_name\_via\_interface\_tag(comp\_id, interface\_tag, comp\_full\_name, annotation)**
* Return value [LOGICAL; OUT]: *true* will be returned only when the corresponding component model full name is successfully obtained.
* comp\_id [INTEGER; IN]: The ID of the component model corresponding to the interface tag.
* interface\_tag [CHARACTER; IN]: The interface tag for the fixed coupling interface. It has the same meaning with the parameter “interface\_tag” of the APIs CCPL\_register\_export\_interface and CCPL\_register\_import\_interface. It has a maximum length of 600 characters.
* comp\_full\_name [CHARACTER; IN]: The output parameter that records the component model full name corresponding to the interface tag.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API tries to get the full name of the component model corresponding to the given interface tag. *true* will be returned when the corresponding model full name is obtained, while false will be returned when failing to find the corresponding model full name.

## CCPL\_connect\_fixed\_interfaces

* **SUBROUTINE CCPL\_connect\_fixed\_interfaces(comp\_full\_name1, comp\_full\_name2, annotation)**
* comp\_full\_name1 [CHARACTER; IN]: The full name of the first component model. It has a maximum length of 512 characters.
* comp\_full\_name2 [CHARACTER; IN]: The full name of the second component model. It has a maximum length of 512 characters.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API connects pairs of fixed coupling interfaces between the given two component models, where the corresponding coupling procedures will be generated automatically. The given two component models can be the same. In other words, C-Coupler2 support that two fixed interfaces are in the same component model. There is no order between the given two component models. At each time of connecting fixed coupling interfaces, any process in any of the given two component models must make a unique call of this API at the same time.

## CCPL\_execute\_interface\_using\_id

* **logical FUNCTION CCPL\_execute\_interface\_using\_id(interface\_id, bypass\_timer, field\_update\_status, annotation)**
* Return value [LOGICAL; OUT]: If the corresponding coupling interface has been executed successfully, *true* will be returned; otherwise (for example, an import interface fail to obtain the coupling field instances required by the corresponding component model), *false* will be returned.
* interface\_id [INTEGER; IN]: The ID of the interface that is to be executed.
* bypass\_timer [LOGICAL; IN]: A mark used to specify whether bypassing the timer of the coupling interface.
* field\_update\_status [INTEGER, DIMENSION(:), OPTIONAL; OUT]: An array where each element is used to record the status that whether the corresponding output field instance is updated in the current call of this API. Value of *1* means that the corresponding output field instance is updated while value of *0* means that the corresponding output field instance is not updated.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API executes the given coupling interface, and returns true when the execution is successful. A coupling interface can be executed only when its coupling procedures haven been generated. All processes in the corresponding coupling interface are required to call this interface to execute the same coupling interface at the same time. When “bypass\_timer” is set to *true*, the timer will not be considered when executing the coupling interface, which means that the coupling interface will be constantly executed (the field instances will be exported, imported or remapped). Please note that a coupling interface can bypass its timer only at the initialization stage (the time step has not been advanced) of an initial run or hybrid run of the coupled model, and the execution of two coupling interfaces that have connections must bypass or not bypass the timers at the same time. When “bypass\_timer” is set to *false*, the coupling interface will be executed only when its timer is on, and the given interface will be truly executed at most once in the same time step (in other words, for multiple calls of this API that execute the same interface at the same time step, only the first call will be truly executed, while the remaining calls will be bypassed).

## CCPL\_execute\_interface\_using\_name

* **SUBROUTINE CCPL\_execute\_interface\_using\_name(component\_id, interface\_name, bypass\_timer, synch\_mark, annotation)**
* Return value [LOGICAL; OUT]: If the corresponding coupling interface has been executed successfully, *true* will be returned; otherwise (for example, an import interface fail to obtain the coupling field instances required by the corresponding component model), *false* will be returned.
* component\_id [INTEGER; IN]: The ID of the component model corresponding to this interface.
* interface\_name [CHARACTER; IN]: The name of the new instance. It has a maximum length of 80 characters. Each character must be ‘A’~‘Z’, ‘a’~‘z’, 0~9 or ‘\_’.
* bypass\_timer [LOGICAL; IN]: A mark used to specify whether bypassing the timer of the coupling interface. When “bypass\_timer” is set to *true*, the timer will not be considered when executing the coupling interface, which means that the coupling interface will be constantly executed (the field instances will be exported, imported or remapped). When “bypass\_timer” is set to *false*, the coupling interface will be executed only when its timer is on. Please note that a coupling interface can bypass its timer only at the initialization stage of an initial run or hybrid run of the coupled model.
* field\_update\_status [INTEGER, DIMENSION(:), OPTIONAL; OUT]: An array where each element is used to return the status that whether the corresponding output field instance is updated in the current call of this API. Value of 1 means that the corresponding output field instance is updated while value of 0 means that the corresponding output field instance is not updated.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API executes the given coupling interface, and returns true when the execution is successful. A coupling interface can be executed only when its coupling procedures haven been generated. All processes in the corresponding coupling interface are required to call this interface to execute the same coupling interface at the same time. When “bypass\_timer” is set to *true*, the timer will not be considered when executing the coupling interface, which means that the coupling interface will be constantly executed (the field instances will be exported, imported or remapped). Please note that a coupling interface can bypass its timer only at the initialization stage (time step has not been advanced) of an initial run or hybrid run of the coupled model, and the execution of two coupling interfaces that have connections must bypass or not bypass the timers at the same time. When “bypass\_timer” is set to *false*, the coupling interface will be executed only when its timer is on, and the given interface will be truly executed at most once in the same time step (in other words, for multiple calls of this API that execute the same interface at the same time step, only the first call will be truly executed, while the remaining calls will be bypassed).

## API for restart management

Similar to models, C-Coupler2 should also have the ability of restarting model simulation. C-Coupler2 provides two APIs according, as listed out in the following Table X. Currently, the restart management of C-Coupler2 only serves the local variables or data managed by C-Coupler2. In other words, the restart ability of component model variables (including the component model fields that have been registered to C-Coupler2 as coupling field instances) is still required to be achieved by models themselves. To achieve restart capability for a coupled system that using C-Coupler2 for model coupling, all active component models should separately call the two APIs CCPL\_do\_restart\_write and CCPL\_do\_restart\_read. Besides “initial” run, C-Coupler2 supports three types of model run: “continue”, “branch” and “hybrid”, which are related to the restart capability.

Table X APIs for coupling interface management.

|  |  |  |
| --- | --- | --- |
| No. | API | Brief description |
| 1 | CCPL\_do\_restart\_write | Write local variables or data of C-Coupler2 into data file |
| 2 | CCPL\_do\_restart\_read | Read in local variables or data of C-Coupler2 from a restart data file |
| 3 | CCPL\_is\_restart\_timer\_on | Check whether the implicit restart timer on |

## CCPL\_do\_restart\_write

* **SUBROUTINE CCPL\_do\_restart\_write(comp\_id, bypass\_timer, annotation)**
* comp\_id [INTEGER; IN]: The ID of the given component model.
* bypass\_timer [LOGICAL; IN]: a mark used to specify whether bypassing the implicit restart timer of the given component model when writing the data for restart capability.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API writes local variables or data of C-Coupler2 into a data file for the given component model. The restart data files will be put under the data directory of the given component model (please refer to Section XXX). All processes of the given component model are required call this API at the same time. If “bypass\_timer” is set to true, restart writing happens when this API is called; otherwise, the implicit restart timer of the given component model will control the time of restart writing, which means that restart writing will not truly happen when the implicit restart timer is not on. Please note that, currently C-Coupler2 forces all component models share the same implicit restart timer, which is specified by the configuration file “CCPL\_dir/config/all/env\_run.xml” (please refer to section X for details).

## CCPL\_do\_restart\_read

* **SUBROUTINE CCPL\_do\_restart\_read(comp\_id, specified\_restart\_file, annotation)**
* comp\_id [INTEGER; IN]: The ID of the given component model.
* specified\_restart\_file [CHARACTER, OPTIONAL; IN]: a specific file used for reading in restart data.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API restarts C-Coupler2 through reading in the restart data from an implicit restart data file or an explicit data file specified by the parameter “specified\_restart\_file”. When “specified\_restart\_file” is not specified, the implicit restart file will be determined by the type of model run (“initial”, “continue”, “branch” and “hybrid”; please refer to Section XXX for details). The restart data files should have already been put under the data directory of the given component model (please refer to Section XXX). All processes of the given component model are required to call this API at the same time.

## CCPL\_is\_restart\_timer\_on

* **logical FUNCTION CCPL\_is\_restart\_timer\_on(comp\_id, annotation)**
* comp\_id [INTEGER; IN]: The ID of the given component model.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API checks whether the implicit restart timer of the given component model is on at the current step.

## API for parallel debugging

## CCPL\_report\_log

* **SUBROUTINE CCPL\_report\_log(comp\_id, condition, report\_string, annotation)**
* comp\_id [INTEGER; IN]: The ID of the given component model.
* condition[LOGICAL; IN]: the log will be reported only when “condition” is set to *true*.
* report\_string [CHARACTER; IN]: the log to be written into the log files. It has a maximum length of 512 characters.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API reports a log for a given component model. Please refer to Section XXX for the details that how C-Coupler2 supports parallel debugging.

## CCPL\_report\_progress

* **SUBROUTINE CCPL\_report\_progress(comp\_id, condition, report\_string, annotation)**
* comp\_id [INTEGER; IN]: The ID of the given component model.
* condition[LOGICAL; IN]: the progress log will be reported only when “condition” is set to *true*.
* report\_string [CHARACTER; IN]: the progress log to be written into the log files. It has a maximum length of 512 characters.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API reports a progress log for a given component model. Please refer to Section XXX for the details that how C-Coupler2 supports parallel debugging.

## CCPL\_report\_error

* **SUBROUTINE CCPL\_report\_error(comp\_id, condition, report\_string, annotation)**
* comp\_id [INTEGER; IN]: The ID of the given component model.
* condition[LOGICAL; IN]: the error log will be reported and the model run will be ended only when “condition” is set to *false*.
* report\_string [CHARACTER; IN]: the progress log to be written into the log files. It has a maximum length of 512 characters.
* annotation [CHARACTER, OPTIONAL; IN]: The annotation used to mark the corresponding model code that calls this API. It has a maximum length of 512 characters.
* **Description of this API**

This API reports an error log for a given component model. Please refer to Section XXX for the details that how C-Coupler2 supports parallel debugging.

## Examples of constructing a coupled system with C-Coupler2 APIs

Figure XXX shows an example of how to use C-Coupler2 to specify coupling configuration for a coupled system. Firstly, the two component models *comp1* and *comp2* that cover all processes (process 0~34) and do not share any process, call the API CCP\_register\_component at the same time, to register themselves as the root component models. *Comp1* has two children *comp3* and *comp4* that share processes (process 9~12) between them. All processes of *comp4* first register *comp4* as a child of *comp1*, next set the unique time step, register model grids, register a parallel decomposition, register coupling field instances, specify a coupling field instance as the dynamic surface field of a 3-D grid, define timers and register coupling interfaces. All processes of *comp3* also conduct coupling configuration, similar to *comp4*. As *comp3* and *comp4* share some processes, *comp3* and *comp4* cannot conduct coupling configuration at the same time in most cases (in this example, we specifically make the coupling configuration of *comp3* later than *comp4*), except for the calling of the API CCPL\_connect\_fixed\_interfaces. After each process either in *comp3* orin *comp4* calls the API CCPL\_connect\_fixed\_interfaces simultaneously, *comp4* can execute a fixed coupling interface with the timer bypassed, before it finalizes its coupling configuration stage through the calling the API CCPL\_end\_coupling\_configuration. Similar to *comp4*, *comp3* executes a fixed coupling interface with the timer bypassed and finally finalizes its coupling configuration stage. After both *comp3* and *comp4* finish their coupling configuration stage, *comp1* conduct its coupling configuration, with a flowchart similar to *comp3* and *comp4*. As *comp1* share processes with *comp3* and *comp4*, *comp1* cannot conduct coupling registration simultaneously with *comp3* and *comp4*, and we specifically make the coupling configuration of *comp1* later than *comp3* and *comp4*). As *comp2* does not share any process with *comp1*, *comp3* or *comp4*, *comp2* can conduct coupling registration simultaneously with *comp1*, *comp3* and *comp4*. Finally, *comp1* and *comp2*, the root component models, call the API CCPL\_end\_coupling\_configuration simultaneously to finalize the coupling configuration stage of themselves and the whole coupled system, when the coupling generator is invoked to generate the coupling procedures of the whole coupled system.

Figure YYY shows an example for how to use C-Coupler2 to conduct model coupling in the initialization stage of the coupled system referred in Figure XXX, where we assume that *comp1* and *comp2* are coupled together, *comp3* and *comp4* are coupled together, and *comp3* and *comp4* depend on some boundary condition from *comp1*. In this example, all coupling interfaces are executed with the timer bypassed. As *comp1* and *comp2* do not share any process, they can execute coupling interfaces at the same time for initializing the model execution. Please note that deadlocks may exist between two component models that are two-way coupled, when both of them are executing import interfaces that are waiting for the data from their export interfaces that have not been executed. Deadlocks may be avoided through adjusting the execution order of between export interfaces and import interfaces. As the export interfaces are generally non-blocking in sending out the coupling field instances, we propose to execute an export interface before an import interface when the two interfaces do not have dependency (the export interface does not depend on the data obtained by the import interface). Deadlocks can also be avoided through enlarging the “remote\_lag\_count” (please refer to Section XXX) in the timer of an import interface. At the end of the initialization stage of *comp1* and *comp2*, *comp1* and *comp2* calls the API CCPL\_do\_restart\_read that will import restart data in a continue run, a branch run or a hybrid run but will be bypassed in an initial run. As *comp3* and *comp4* share some processes, they cannot execute coupling interfaces at the same time, and in this example, they execute the coupling interfaces alternately. Please note that when the two component models that are coupled with each other share some processes, it will be of higher possibility to encounter a deadlock.

Figure ZZZ shows an example of model coupling in the kernel (time integration) stage of the coupled system referred in Figure XXX and Figure YYY, where we still use the assumption in Figure YYY and further assume that *comp1* and *comp2* have the same time step that is two times of the time step of *comp3* and *comp4*. In this example, all coupling interfaces are executed without bypassing the timers. In a time step of *comp1* and *comp2*, they execute coupling interfaces at the same time, call API CCPL\_do\_restart\_write to generate restart data file when the restart timer is bypassed or is on, and finally call CCPL\_advance\_time to advance the model time managed by C-Coupler2. We highly propose to check the consistency of model time between a component model and C-Coupler through calling the API CCPL\_check\_current\_time.

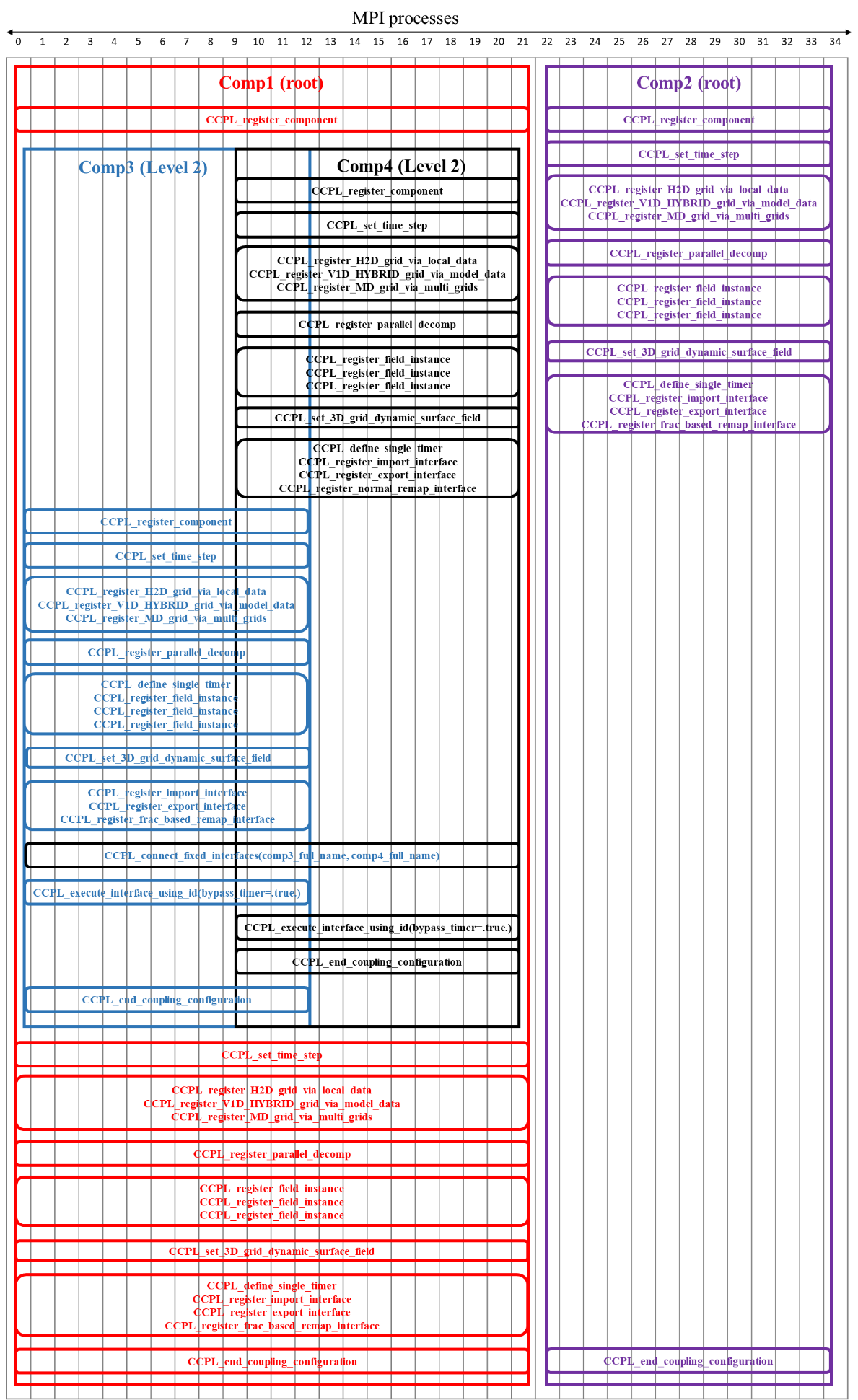


Figure XXX An example of coupling configuration stage of a coupled system constructed with C-Coupler2. The texts and boxes in the same color corresponds to the same component model.

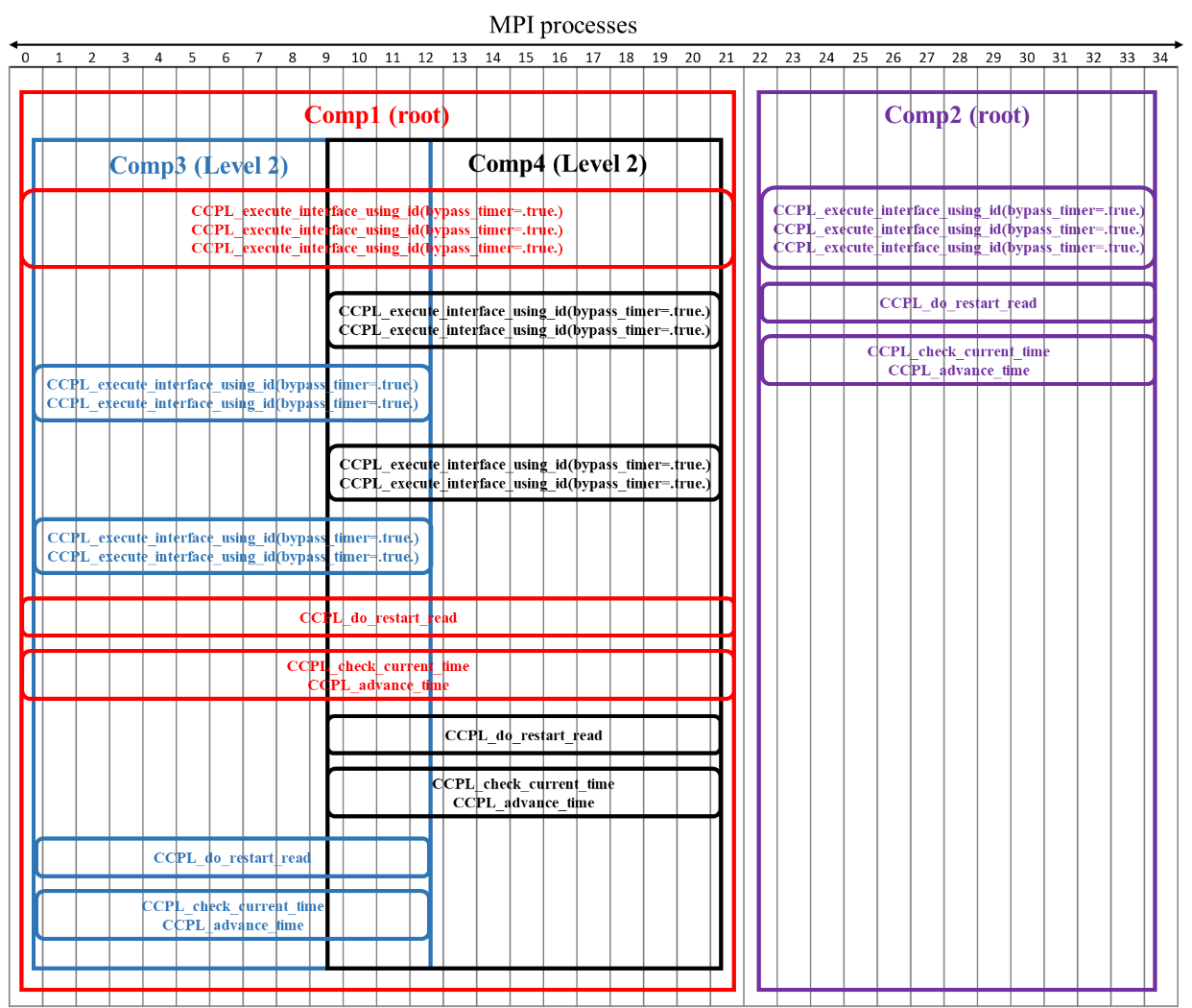


Figure YYY An example of model coupling in the initialization stage of a coupled system constructed with C-Coupler2. The texts and boxes in the same color corresponds to the same component model.

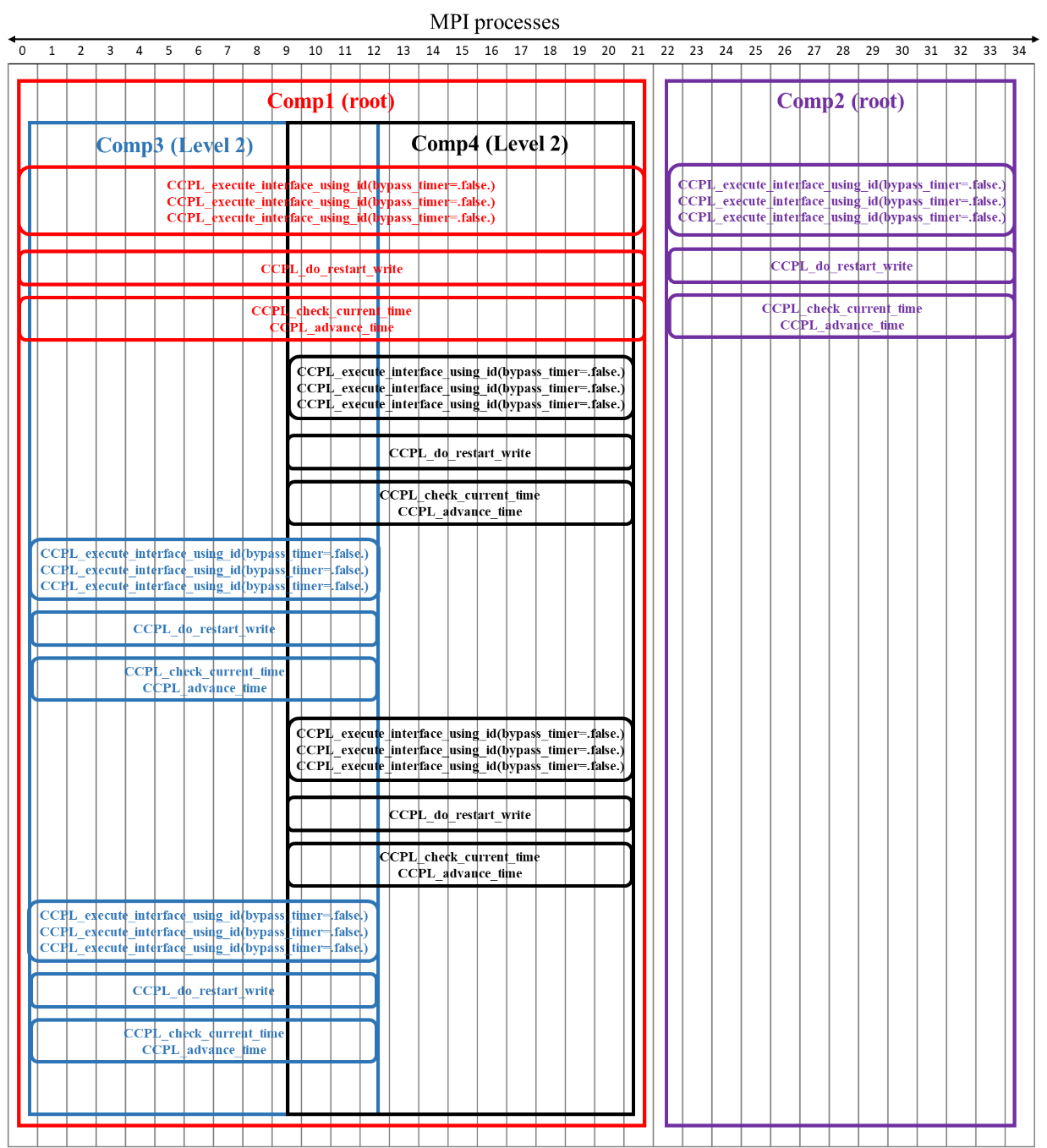


Figure ZZZ An example of model coupling in the kernel (time integration) stage of a coupled system constructed with C-Coupler2. The texts and boxes in the same color corresponds to the same component model.

## Coupling configuration with configuration files

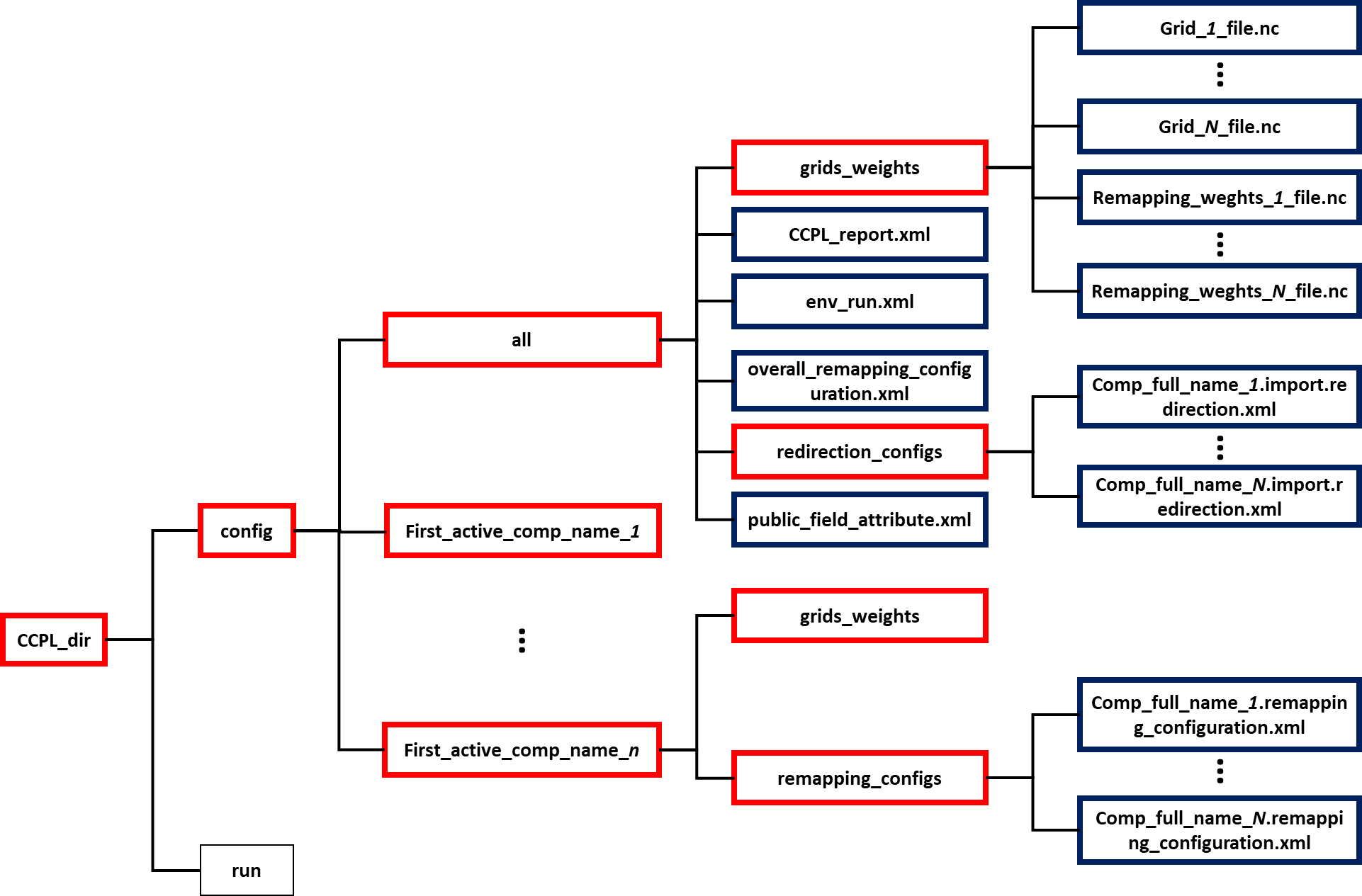


Figure XXX Directory of the configuration files of C-Coupler2. A red-edged box stands for a directory and a blue-edged box stands for a specific file.

For the coupling configuration that cannot be specified through the C-Coupler2 APIs, C-Coupler2 provides a set of configuration files accordingly. A configuration file is either a text file in XML format or a data file in NETCDF format. As shown in Figure XXX, all configuration files are put under the directory “CCPL\_dir/config”. The configuration files that are shared by the whole coupled model are put under the directory “CCPL\_dir/config/all”. The configuration files specific to component models are put under a directory “CCPL\_dir/config/First\_active\_comp\_name­\_*N*”, where “First\_active\_comp\_name­\_*N*” stands for the name of an active component model that is the first one registered in a subset of processes, and the configuration files corresponding to the component models that are registered in the same subset of processes are put under this directory. Both of the directories “CCPL\_dir/config/all” and “CCPL\_dir/config/First\_active\_comp\_name­\_*N*” contain a subdirectory “grids\_weights” that can store a set of grid data files for grid registration and a set of remapping weight files for remapping coupling fields. The “CCPL\_dir/config/First\_active\_comp\_name­\_*N*” also contains a subdirectory “remapping\_configs” that stores a set of XML files for specifying how to remap coupling fields.

Table XXX briefly describes each kind of configuration files. The grid data file “grid\_*N­­*\_file.nc” has already been introduced in section XXX. In the following context of this chapter, each of remaining configuration files will be introduced. As “overall\_remapping\_configuration.xml”, “comp\_full\_name\_*N*.remapping\_configuration.xml” and “remapping\_weights\_*N­*\_file.nc” are used for describing the configuration of remapping, they will be introduced together.

Table XXX Configuration files provided by C-Coupler2

|  |  |  |
| --- | --- | --- |
| File name | Format | Description |
| env\_run.xml | XML | Used to specify the parameters to run a simulation of the coupled model |
| public\_field\_attribute.xml | XML | Used to specify the attributes of each coupling field |
| CCPL\_report.xml | XML | Used to enable or disable the log report and error check by C-Coupler2 |
| overall\_remapping\_configuration.xml | XML | Used to specify how to remap coupling fields for the whole coupled model |
| comp\_full\_name\_*N*.remapping\_configuration.xml | XML | Used to specify how to remap coupling fields for a subset of component models |
| remapping\_weights\_*N­*\_file.nc | NETCDF | A data file with remapping weights from one grid to another |
| comp\_full\_name\_*N*.import.redirection.xml | XML | Used to specify coupling connections for some import interfaces of the corresponding component model |
| grid\_*N­­*\_file.nc | NETCDF | A data file with grid data |

## env\_run.xml

<?xml version="1.0" ?>

<Time\_setting

case\_name="C-Coupler2 testing"

model\_name="ideal\_model\_for\_CCPL2"

run\_type="initial"

leap\_year="on"

start\_date="00040331"

start\_second="0"

rest\_freq\_unit="seconds"

rest\_freq\_count="14400"

rest\_ref\_case="C-Coupler testing"

rest\_ref\_date="00040401"

rest\_ref\_second="0"

stop\_option="nhours"

stop\_date="00010101"

stop\_second="0"

stop\_n="30"

/>

Figure XXX An example of the configuration file “env\_run.xml”

As shown in Figure XXX, the configuration file “env\_run.xml” enables users to specify a series of global attributes for the simulation run of the whole coupled model. These global attributes are described as follows:

* case\_name [CHARACTER]: The name of the simulation run. It has a maximum length of 80 characters.
* case\_description [CHARACTER, OPTIONAL]: The description of the simulation run. It has a maximum length of 1000 characters.
* model\_name [CHARACTER]: The name of the whole coupled model. It has a maximum length of 80 characters.
* run\_type [CHARACTER]: The type to run the simulation. Four types are supported by C-Coupler2: initial run, continue run, branch run and hybrid run. “run\_type” must be set to “initial”, “continue”, “branch” or “hybrid”.
* leap\_year [CHARACTER]: To specify whether leap year is enabled in the simulation run. “leap\_year” must be set to “on” or “off”, where “on” means that leap year is enabled.
* start\_date [INTEGER]: The date to start the simulation run. It must be a positive value in the format of YYYYMMDD, where YYYY means year, MM means month and DD means day.
* start\_second [INTEGER]: The second of start time on the start day. It must be no smaller than 0 and smaller than 86400.
* rest\_freq\_unit [CHARACTER]: The unit of the period of producing restart data files. The unit can be second (“second”, “seconds”, “nsecond” or “nseconds”), day (“day”, “days”, “nday” or “ndays”), month (“month”, “months”, “nmonth” or “nmonths”) or year (“year”, “years”, “nyear” or “years”). It also can be “none” which means that no restart data files will be produced in the simulation run.
* rest\_freq\_count [INTEGER, OPTIONAL]: The count of the period of producing restart data files corresponding to the unit. When “rest\_freq\_unit” is not set to “none”, “rest\_freq\_count” must be set to a positive value.
* rest\_ref\_case [CHARACTER, OPTIONAL]: The name of a reference simulation case whose restart data files will be used for restarting the current simulation run. It must be set when the “run\_type” has been set to “branch” or “hybrid”.
* rest\_ref\_date [INTEGER]: A date that is used to specify a detailed restart data file produced by the reference simulation case. It must be a positive value in the format of YYYYMMDD, where YYYY means year, MM means month and DD means day. It must be set when the “run\_type” has been set to “branch” or “hybrid”.
* rest\_ref\_second [INTEGER]: A second number that is used to specify a detailed restart data file produced by the reference simulation case. It must be no smaller than 0 and smaller than 86400. It must be set when the “run\_type” has been set to “branch” or “hybrid”.
* stop\_option [CHARACTER]: The option for specifying how to stop the simulation run. It can be set to date (“date”), or a time unit such as second (“second”, “seconds”, “nsecond” or “nseconds”), minute (“minute”, “minutes”, “nmunite” or “nmunites”), hour (“hour”, “hours”, “nhour” or “nhours”), day (“day”, “days”, “nday” or “ndays”), month (“month”, “months”, “nmonth”, “nmonths”) and year (“year”, “years”, “nyear” or “nyears”).
* stop\_date [INTEGER, OPTIONAL]: The model date to stop the simulation run. It must be a positive value in the format of YYYYMMDD, where YYYY means year, MM means month and DD means day. It must be set when the “stop\_option” has been set to “date”.
* stop\_second [INTEGER, OPTIONAL]: The second number on the “stop\_date” to stop the simulation run. It must be no smaller than 0 and smaller than 86400. It must be set when the “stop\_option” has been set to “date”.
* stop\_n [INTEGER]: A number that is used to control the time length when the “stop\_option” is set to a time unit. In other words, it must be set when the “stop\_option” has been set to a time unit. It can be a positive value or -999. When it is a negative value -*999*, it indicates that the simulation run is endless.

## public\_field\_attribute.xml

<?xml version="1.0" ?>

<field name="evap" dimensions="H2D" long\_name="water evaporation" type="flux" default\_unit="kg/s/m^2" />

<field name="atm\_t" dimensions="V3D" long\_name="air temperature" type="state" default\_unit="kelvin" />

<field name="ps" dimensions="H2D" long\_name="surface pressure" type="state" default\_unit="Pa" />

<field name="ifrac" dimensions="H2D" long\_name="fraction of sea ice" type="state" default\_unit="unitless" />

<field name="sst" dimensions="H2D" long\_name="sea surface temperature" type="state" default\_unit="kelvin" />

<field name="sss" dimensions="H2D" long\_name="salty surface temperature" type="state" default\_unit="kelvin" />

<field name="tbot" dimensions="H2D" long\_name="bottom atm level temperature" type="state" default\_unit="kelvin" />

<field name="CO2\_avg" dimensions="0D" long\_name="averaged CO2 concentration" type="state" default\_unit="parts per million (ppm)"/>

Figure XXX An example of the configuration file “public\_field\_attribute.xml”

As introduced in Section XXX, when registering a coupling filed instance to C-Coupler2, the corresponding field name should be specified as an input parameter. A field name is legal only when there is a corresponding entry in the configuration file public\_field\_attribute.xml that is shared by all component models in a coupled model. When the coupling generator tries to automatically generate coupling procedures, field names are used to analyze possible coupling connections between coupling interfaces: an import interface and an export interface can have a coupling connection only when their coupling field instances have common field names.

Figure XXX show an example of the configuration file “public\_field\_attribute.xml” where each XML node of “field” specifies a set of attributes corresponding to a distinct field name. These attributes are described as follows:

* name [CHARACTER]: The field name. It has a maximum length of 80 characters.
* dimensions [CHARACTER]: A label of grid dimensions corresponding to the field. It can be “0D”, “H2D”, “V1D”, “V3D”. “0D” means that the field is a scalar variable not on any grid. “H2D” means that the field is on a horizontal grid. “V1D” means that the field is on a vertical grid. “V3D” means that the field is on a 3-D grid that consists of a horizontal grid and vertical grid.
* long\_name [CHARACTER]: The description of the field. It has a maximum length of 1000 characters.
* Type [CHARACTER]: The type of the field: “state” or “flux”.
* Default\_unit [CHARACTER]: The default unit of the field. It has a maximum length of 80 characters.

## CCPL\_report.xml

<?xml version="1.0" ?>

<Report\_setting

report\_internal\_log="off"

report\_external\_log="off"

report\_error="off"

report\_progress="on"

/>

Figure XXX An example of the configuration file “public\_field\_attribute.xml”

C-Coupler2 can report a lot of log information for itself and component models, can conduct a lot of automatic error check during model coupling operations, and enables each process of the coupled model to have its own log file, which of course can facilitate parallel debugging and improve reliability in constructing a coupled model. Considering that log information and automatic error checks should be necessary for constructing a coupled model but would be time-consuming and redundant for a coupled model that is stable enough for usage after a lot of software testing, C-Coupler2 enables users to enable/disable the log information and error checks through the configuration file “public\_field\_attribute.xml”. Figure XXX shows an example of this configuration file. The attributes in this file are described as follows:

* report\_internal\_log [CHARACTER, OPTIONAL]: It is used to specify whether to enable (the value is “on”) or disables (the value is “off”) C-Coupler2 to report its own log information. Its default value is “off”, which means that C-Coupler2 will not report its own log information when the configuration file does not include this attribute.
* report\_external\_log [CHARACTER, OPTIONAL]: It is used to specify whether to enable (the value is “on”) or disables (the value is “off”) C-Coupler2 to report log information of component models. Its default value is “off”, which means that C-Coupler2 will not report log information of component models when the configuration file does not include this attribute.
* report\_error [CHARACTER, OPTIONAL]: It is used to specify whether to enable (the value is “on”) or disables (the value is “off”) C-Coupler2 to conduct time-consuming error check. Its default value is “off”, which means that C-Coupler2 will not conduct time-consuming error check when the configuration file does not include this attribute.
* report\_progress [CHARACTER, OPTIONAL]: It is used to specify whether to enable (the value is “on”) or disables (the value is “off”) C-Coupler2 to report execution progress. Its default value is “off”, which means that C-Coupler2 will not report execution progress when the configuration file does not include this attribute. Please note that, only the root process (ID is 0) of a component model can report execution progress.

## Remapping configuration

C-Coupler2 provides the function of remapping configuration that enables users to flexibly specify how to remap coupling fields between grids:

1. For the remapping from a source horizontal grid to a target horizontal grid, users can either use the remapping weights that are automatically generated by C-Coupler2 in parallel, or use the remapping weights that are read from an existing remapping weight file produced by the software such as SCRIP, ESMF, CoR, etc.
2. Different coupling fields can have different remapping configurations in the same component model, and the same coupling field can have different remapping configurations in different component models.
3. Given a coupling field, a component model can use its own remapping configuration specified in the corresponding configuration file “*comp\_full\_name*.remapping\_configuration.xml” ((Figure XXX and Table XXX), or use the inherited remapping configuration that is used by its parent when its own remapping configuration is not specified. A root component model that does not have a parent can use the remapping configuration specified in the configuration file “overall\_remapping\_configuration.xml” (Figure XXX and Table XXX) or can use the default remapping configuration set by C-Coupler2 when “overall\_remapping\_configuration.xml” is not specified. In the default remapping configuration, the bilinear remapping algorithm is used for remapping “state” fields between horizontal grids, the conservative remapping algorithm is used for remapping “flux” fields between horizontal grids, and the linear remapping algorithm is used for remapping in both the vertical dimension and the time dimension. Please note that all remapping weights in the default remapping configuration are generated by C-Coupler2 automatically.
4. The configuration file “\*remapping\_configuration.xml” enables users to specify the remapping configuration for all coupling fields, the coupling fields with the same type (“flux” or “state”), or a set of coupling fields (even only one field) at each time, and enables that an exported coupling field of an component model can have its own remapping configuration.

In the following context of this subsection, we will further introduce the remapping weight file, the priority based hierarchy scheme and the configuration file.

## Remapping weight file

The remapping weight file should be in a NETCDF format oriented from the remapping weight file produced by the software SCRIP. The software such as ESMF and CoR can also produce such format of remapping weight files. Figure XXX shows an example of this format, with the variables necessary to C-Coupler2. A remapping weight file with these variables can be used by C-Coupler2, no matter which software generates the remapping weight file. These variables are briefly described in Table XXX.

Table XXX Descriptions of the variables required by C-Coupler2 in a remapping weight file

|  |  |
| --- | --- |
| Variable | description |
| yc\_a | Latitude (Y) value of the center of each cell of the source grid |
| xc\_a | Longitude (X) value of the center of each cell of the source grid |
| yc\_b | Latitude (Y) value of the center of each cell of the target grid |
| xc\_b | Longitude (X) value of the center of each cell of the target grid |
| yv\_a | Latitude (Y) value of the vertexes of each cell of the source grid |
| xv\_a | Longitude (X) value of the vertexes of each cell of the source grid |
| yv\_b | Latitude (Y) value of the vertexes of each cell of the target grid |
| xv\_b | Longitude (X) value of the vertexes of each cell of the target grid |
| mask\_a | Mask of each cell of the source grid |
| mask\_b | Mask of each cell of the target grid |
| area\_a | Area of each cell of the source grid |
| area\_b | Area of each cell of the target grid |
| col | Global cell index in the source grid in each remapping weight |
| row | Global cell index in the target grid in each remapping weight |
| S | Weight value in each remapping weight |

dimensions:

n\_a = 70560 ;

n\_b = 7680 ;

nv\_a = 4 ;

nv\_b = 4 ;

n\_s = 84988 ;

variables:

double yc\_a(n\_a) ;

yc\_a:units = "degrees" ;

double yc\_b(n\_b) ;

yc\_b:units = "degrees" ;

double xc\_a(n\_a) ;

xc\_a:units = "degrees" ;

double xc\_b(n\_b) ;

xc\_b:units = "degrees" ;

double yv\_a(n\_a, nv\_a) ;

yv\_a:units = "degrees" ;

double xv\_a(n\_a, nv\_a) ;

xv\_a:units = "degrees" ;

double yv\_b(n\_b, nv\_b) ;

yv\_b:units = "degrees" ;

double xv\_b(n\_b, nv\_b) ;

xv\_b:units = "degrees" ;

int mask\_a(n\_a) ;

mask\_a:units = "unitless" ;

int mask\_b(n\_b) ;

mask\_b:units = "unitless" ;

double area\_a(n\_a) ;

area\_a:units = "square radians" ;

double area\_b(n\_b) ;

area\_b:units = "square radians" ;

int col(n\_s) ;

int row(n\_s) ;

double S(n\_s) ;

Figure XXX An example of the NETCDF format of the remapping weight file

## Configuration file “\*remapping\_configuration.xml”

“\*remapping\_configuration.xml” is an XML formatted configuration file that enables users to specify the rules of how to remap the coupling fields (for example, Figure YYY). Its unique XML node “root” contains a set of active XML nodes of “remapping\_setting” (for example, L1 to L15, L16 to L27, and L28 to L39 in Figure YYY), each of which specifies a rule of how to remap some coupling fields. An XML node of “remapping\_setting” consists of two active XML nodes: an active XML node of “remapping\_algorithms” (for example, L2 to L13, L17 to L23, and L29 to L34 in Figure YYY) that is about the algorithms or weight files for remapping, and an active XML node of “fields” (for example, L14, L24 to L26, L35 to L38 in Figure YYY) that is about the coupling fields the XML node of “remapping\_setting” is for. An XML node of “remapping\_setting” is active only when its attribute of “status” is set to “on” (for example, L1, L16, and L28 in Figure YYY), and will be neglected when its attribute of “status” is set to “off”.

* **XML node of “remapping\_algorithms”**

An XML node of “remapping\_algorithms” can include at most one active XML node of “H2D\_algorithm” (for example, L3 to L5, and L18 in Figure YYY), at most one active XML node of “V1D\_algorithm” (for example, L6 to L8, and L30 to L33 in Figure YYY), and at most one active XML node of “H2D\_weights” (for example, L9 to L12, and L19 to L22 in Figure YYY). An XML node of “remapping\_algorithms” is active only when its attribute of “status” is set to “on” (for example, L3, and L18 in Figure YYY), and will be neglected when its attribute of “status” is set to “off”.

* + **XML node of “H2D\_algorithm”**

An XML node of “H2D\_algorithm” is used to specify a remapping algorithm for data interpolation between two horizontal grids, where the remapping algorithm is specified through the attribute of “name” (for example, L3, and L18 in Figure YYY) and the algorithm parameters can be further set in this XML node (for example, L4 in Figure YYY). An XML node of “H2D\_algorithm” is active only when its attribute of “status” is set to “on” (for example, L3, and L18 in Figure YYY), and will be neglected when its attribute of “status” is set to “off”.

* + **XML node of “V1D\_algorithm”**

An XML node of “V1D\_algorithm” is similar to an XML node of “H2D\_algorithm”, but for data interpolation between two vertical grids (for example, L6 to L8, and L30 to L33 in Figure YYY).

* + **XML node of “H2D\_weights”**

An XML node of “H2D\_weights” enables users to specify a set of external remapping weight files each of which is for data interpolation between two specific horizontal grids (for example, L9 to L12, and L19 to L22 in Figure YYY). A remapping weight file must be a NETCDF file following the format of remapping weight file produced by the remapping software SCRIP (called SCRIP format hereafter). Other remapping software such as CoR can also produce SCRIP formatted remapping weight file. An XML node of “H2D\_weights” is active only when its attribute of “status” is set to “on” (for example, L9, and L19 in Figure YYY), and will be neglected when its attribute of “status” is set to “off”.

An XML node of “remapping\_algorithms” can include an active XML node of “H2D\_algorithm” and an active XML node of “H2D\_weights” at the same time while “H2D\_weights” has a higher priority than “H2D\_algorithm”. When both a specified remapping weight file and the specified remapping algorithm can be used for handling the data interpolation between two specific horizontal grids, only the specified remapping weight file will be used.

* **XML node of “fields”**

XML nodes of “fields” enable users to specify different remapping settings for different subsets of coupling fields, so as to improve the flexibility of remapping configuration. In an XML node of “fields”, users can determine how to specify coupling fields, through setting the attribute of “specification” to “name” (for example, L35 in Figure YYY), “type” (for example, L24 in Figure YYY), or “default” (for example, L14 in Figure YYY). When the attribute of “specification” is set to “name”, the name of the coupling fields must be listed out in the XML node of “fields” (for example, the field names of “t\_atm\_3D” and “ghs\_atm\_3D” in Figure YYY). When the attribute of “specification” is set to “type”, only one type of coupling fields should be specified in the XML node of “fields” (for example, L25 in Figure YYY). Currently, there are only two types of coupling fields: “state” and “flux”, and the type of each field is set in the *public\_field\_attribute.xml* (in Section XXX). When the attribute of “specification” is set to “default”, no more information is required (for example, L25 in Figure YYY), and it means any coupling field can use the corresponding remapping setting.

An XML node of “fields” is active only when its attribute of “status” is set to “on” (for example, L14, L24, and L35 to L38 in Figure YYY), and will be neglected when its attribute of “status” is set to “off”.

* + **Constraints and rules to use “\*remapping\_configuration.xml”**

To use “\*remapping\_configuration.xml” for remapping configuration, users should note the following constraints and rules:

1. When multiple active XML nodes of “remapping\_setting” are included, there must be no conflicts of fields setting between these XML nodes of “remapping\_setting”.
   1. There can be at most one active XML node of “remapping\_setting” where the attribute of “specification” in the corresponding XML node of “fields” is set to “default”.
   2. There can be at most two active XML nodes of “remapping\_setting” where the attributes of “specification” in the corresponding XML nodes of “fields” are both set to “type” and the field types in these two XML nodes must be different, because currently C-Coupler only support two types of fields (“state” and “flux”).
   3. There can be multiple active XML nodes of “remapping\_setting” where the attributes of “specification” in the corresponding XML nodes of “fields” are both set to “name”, while there must be no common field name among these XML nodes.
2. Given a coupling field, multiple active XML nodes of “remapping\_setting” can have different priorities in determining the specific remapping setting for the coupling field: the active XML node with the name of the coupling field specified has the highest priority; the active XML node with the type of the coupling field specified has a medium priority; the active XML node for “default” (the corresponding attribute of “specification” is set to “default”) has the lowest priority.
3. It is possible that the remapping setting 3-D coupling field is determined by two active XML nodes of “remapping\_setting”, where one active XML node determines how to remap between horizontal grids and the other determines how to remap between horizontal grids.

<root>

L1: <remapping\_setting status="on">

L2: <remapping\_algorithms status="on">

L3: <H2D\_algorithm status="on" name="bilinear">

L4: <parameter name="enable\_extrapolate" value="true" />

L5: </H2D\_algorithm>

L6: <V1D\_algorithm status="on" name="linear">

L7: <parameter name="enable\_extrapolate" value="true" />

L8: </V1D\_algorithm>

L9: <H2D\_weights status="on">

L10: <file name="map\_to\_global\_grid1\_default.nc" />

L11: <file name="map\_to\_regional\_grid1\_default.nc" />

L12: </H2D\_weights>

L13: </remapping\_algorithms>

L14: <fields status="on" specification="default" />

L15: </remapping\_setting>

L16: <remapping\_setting status="on">

L17: <remapping\_algorithms status="on">

L18: <H2D\_algorithm status="on" name="conserv\_2D" />

L19: <H2D\_weights status="on">

L20: <file name="map\_to\_global\_grid1\_conserv.nc" />

L21: <file name="map\_to\_regional\_grid1\_conserv.nc" />

L22: </H2D\_weights>

L23: </remapping\_algorithms>

L24: <fields status="on" specification="type">

L25: <entry value="flux" />

L26: </fields>

L27: </remapping\_setting>

L28: <remapping\_setting status="on">

L29: <remapping\_algorithms status="on">

L30: <V1D\_algorithm status="on" name="linear">

L31: <parameter name="enable\_extrapolate" value="true" />

L32: <parameter name="use\_logarithmic\_coordinate" value="true" />

L33: </V1D\_algorithm>

L34: </remapping\_algorithms>

L35: <fields status="on" specification="name">

L36: <entry value="t\_atm\_3D" />

L37: <entry value="ghs\_atm\_3D" />

L38: </fields>

L39: </remapping\_setting>

</root>

Figure YYY An example about “\*remapping\_configuration.xml”

## A priority based hierarchical scheme

## *Comp\_full\_name*.import.redirection.xml

*Comp\_full\_name*.import.redirection.xml is an XML formatted configuration file corresponding to the component model with the full name of *Comp\_full\_name*, which enables users flexibly set the sources for the input coupling fields and input horizontal grids (for example, Figure XXX). It consists of an XML node of “component\_import\_interfaces\_configuration” that specifies the sources for the coupling fields of dynamic import interfaces (registered to C-Coupler through the API “xxx”) (for example, L1 to L36 in Figure XXX), and an XML node of “component\_grid\_redirection\_configuration” that specifies the sources for the horizontal grids that will be registered based on the horizontal grids from another component model (registered to C-Coupler through the API “XXX”) (for example, L37 to L40 in Figure XXX).

## XML node of “component\_import\_interfaces\_configuration”

The XML node of “component\_import\_interfaces\_configuration” consists of a set of XML nodes of “import\_interface”, each of which specifies the sources for coupling fields of a dynamic import interface (for example, L2 to L11 in Figure XXX is for the import interface named “receive\_from\_OCN”, and L12 to L35 is for the import interface named “receive\_from\_ATM”). An XML node of “import\_interface” consists of a set of XML nodes of “import\_redirection”, each of which corresponds to a subset of coupling fields (for example, L13 to L21, L22 to L30, and L31 to L34 in Figure XXX are the XML nodes of “import\_redirection” for the import interface named “receive\_from\_ATM”). An XML node of “import\_redirection” consists of an active XML node of “fields” (for example, L4, L14 to L17, L23 to L26, and L32 in Figure XXX) and an active XML node of “components” (for example, L5 to L7, L18 to L20, L27 to L29, and L33 in Figure XXX). The XML node of “fields” specifies the coupling fields regarding the corresponding “import\_redirection”, and the XML node of “components” specifies the component models that provide the coupling fields that are specified in the XML node of “fields”.

An XML node of “import\_interface” is active only when its attribute of “status” is set to “on” (for example, L1, L16, and L28 in Figure YYY), and will be neglected when its attribute of “status” is set to “off”. Similarly, an XML node of “import\_redirection” also has such an attribute of “status” (for example, L3, L9, L13, L22, and L31 in Figure XXX).

* **XML node of “fields”**

In an XML node of “fields”, there is an attribute of “default” with the value of “off”, “remain” or “all” (for example, L4, L14, L23, and L32 in Figure XXX). When the value is “off”, the names of coupling fields are required be added into the XML node of “fields” (for example, L15 to L16, and L24 to L25 in Figure XXX). When the value is “remain” or “all”, no field name can be added into the XML node of “fields” (for example, L4, and L32 in Figure XXX). The value of “remain” means all the remaining coupling fields that have not been referred in the previous XML nodes of “import\_redirection” for the same import interface. The value of “all” means all coupling fields of the corresponding import interface.

An XML node of “fields” is active only when its attribute of “status” is set to “on” (for example, L4, L14, L23, and L32 in Figure XXX), and will be neglected when its attribute of “status” is set to “off”.

* **XML node of “components”**

In an XML node of “components”, there is also an attribute of “default” with the value of “off” or “all” (for example, L5, L18, L27 and L33 in Figure XXX). If the value is set to “off”, detailed information of the component models that will provide the corresponding coupling field instances are required to be added into the XML node of “components” (for example, L5 to L7, L18 to L20, and L27 to L29 in Figure XXX); otherwise, no full name of component models can be added into the XML node of “components” (for example, L33 in Figure XXX). Regarding the detailed information of a component model, users can only provide the full name of the component model, or additionally specify the name of an export interface registered in the component model (for example, L6, and L19 in Figure XXX). The value of “all” means all component models in the coupled model.

An XML node of “components” is active only when its attribute of “status” is set to “on” (for example, L5, L18, L27, and L33 in Figure XXX), and will be neglected when its attribute of “status” is set to “off”.

Figure XXX An example about Comp\_full\_name.import.redirection.xml

<root>

L1: <component\_import\_interfaces\_configuration>

L2: <import\_interface name="receive\_from\_OCN" status="on">

L3: <import\_redirection status="on">

L4: <fields status="on" default="all" />

L5: <components status="on" default="off">

L6: <component comp\_full\_name="ocn\_unique" interface\_name="send\_data\_to\_CPL" />

L7: </components>

L8: </import\_redirection>

L9: <import\_redirection status="off">

…

L10: </import\_redirection>

L11: </import\_interface>

L12: <import\_interface name="receive\_from\_ATM" status="on">

L13: <import\_redirection status="on">

L14: < fields status="on" default="off">

L15: <field name="prec" />

L16: <field name="lwdn" />

L17: </fields>

L18: < components status="on" default="off">

L19: <component comp\_full\_name="atm\_global@atm\_nest\_1" />

L20: </components>

L21: </import\_redirection>

L22: <import\_redirection status="on">

L23: < fields status="on" default="off">

L24: <field name="u" />

L25: <field name="v" />

L26: </fields>

L27: < components status="on" default="off">

L28: <component comp\_full\_name="atm\_global@atm\_nest\_2" />

L29: </components>

L30: </import\_redirection>

L31: <import\_redirection status="on">

L32: < fields status="on" default="remain" />

L33: < components status="on" default="all" />

L34: </import\_redirection>

L35: </import\_interface>

L36: </component\_import\_interfaces\_configuration>

L37: <component\_grid\_redirection\_configuration>

L38: <entry local\_grid\_name="atm\_global\_grid" another\_comp\_full\_name="atm\_global" another\_comp\_grid\_name="H2D\_grid" />

L39: <entry local\_grid\_name="atm\_nest1\_grid" another\_comp\_full\_name=" atm\_global@atm\_nest\_1" another\_comp\_grid\_name="H2D\_grid" />

L40: </component\_grid\_redirection\_configuration>

</root>

* **XML node of “import\_interface”**

An XML node of “import\_interface” can include several XML nodes of “import\_redirection”, each of which corresponds to a separate subset of coupling fields of the corresponding import interface. In other words, different XML nodes of “import\_redirection” cannot share any coupling fields. An XML node of “import\_interface” can only include one XML node of “import\_redirection” when the attribute of “default” in the corresponding XML node of “fields” is set to “all” (for example, L2 to L11 in Figure XXX). There can be several XML nodes of “import\_redirection” where the attribute of “default” in the corresponding XML nodes of “fields” are all set to “off”, optionally following an XML node of “import\_redirection” where the attribute of “default” in the corresponding XML node of “fields” is set to “remain” (for example, L13 to L34 in Figure XXX).

For a brief summary, a valid XML node of “import\_interface” must obey the following constraints:

1. The source of a coupling field can be specified only once.
2. A coupling field can only have one source (among the component models specified in an XML node of “components”, only one component model can provide the corresponding coupling field).

## XML node of “component\_grid\_redirection\_configuration”

The XML node of “component\_grid\_redirection\_configuration” includes a set of entries each of which corresponds to a horizontal grid that has been registered through the API “CCPL\_register\_H2D\_grid\_from\_another\_component” (for example, L37 to L40 in Figure XXX). In an entry, the attribute of “local\_grid\_name” specifies the name of the horizontal grid, which is the parameter “grid\_name” when calling “CCPL\_register\_H2D\_grid\_from\_another\_component”; the attribute of “another\_comp\_full\_name” specifies the full name of the another component model and the attribute of “another\_comp\_grid\_name” specifies the name of the corresponding horizontal grid registered in the another component model.

## Outputs from C-Coupler2

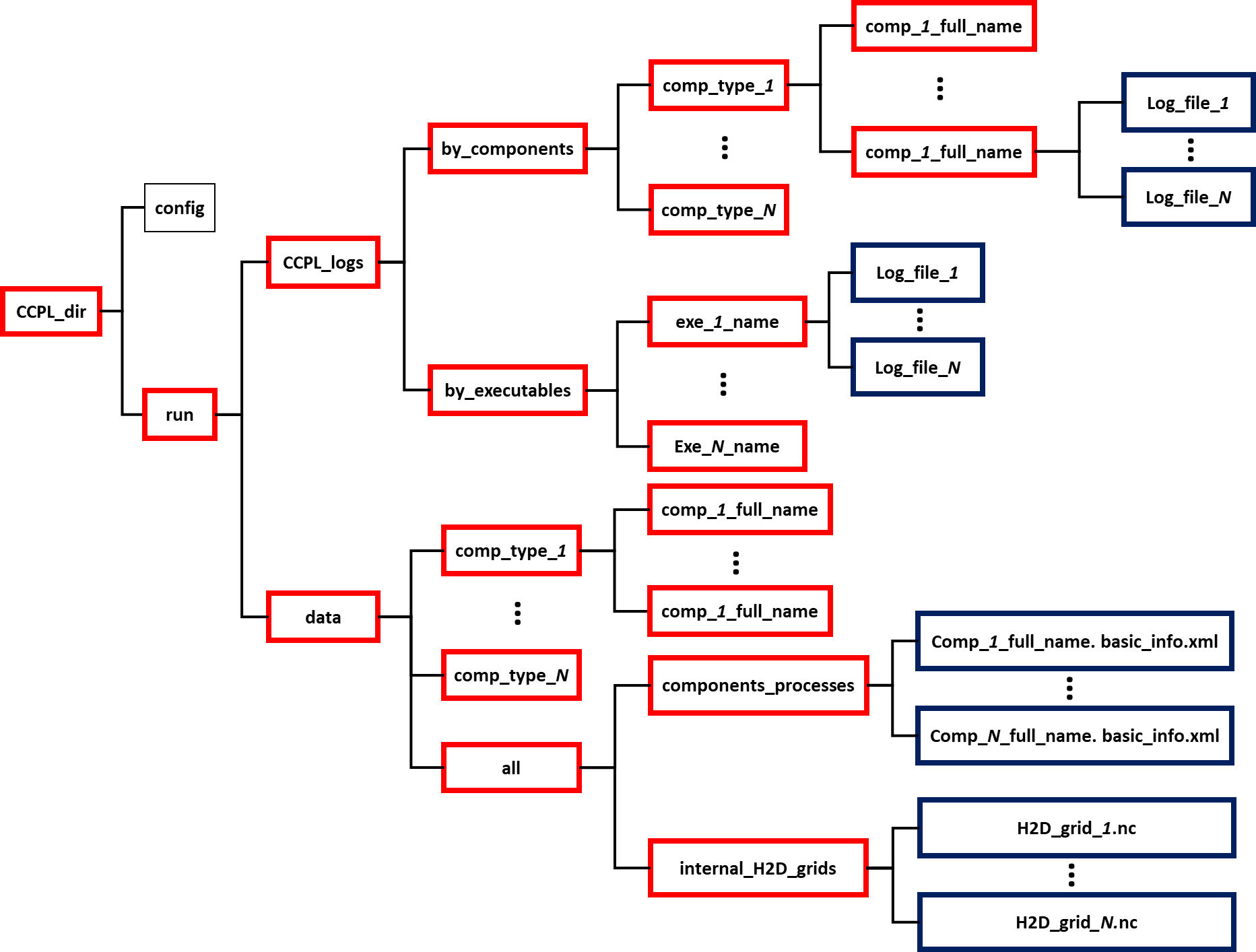


Figure XXX Directories for the outputs from C-Coupler2

As shown in Figure XXX, all outputs from C-Coupler2 are put under the directory “CCPL\_dir/run”, where the sub directory “CCPL\_dir/run/CCPL\_logs” is for the log files for debugging C-Coupler2 and the whole coupled system and the sub directory “CCPL\_dir/run/data” is for the output data files.