# Chapter 8

# Program conventions and techniques

The following items are discussed in this chapter.

- The COHERENS V2.0 code is written in standard FORTRAN 90 format. To improve portability and transparancy a number of programming conventions, further denoted as the "COHERENS conventions", have been adopted. They are described in Section 8.1. These rules are of importance for developers who are working on new developments and have the intention to make the new code available to the COHERENS community.

- Implementation of specific FORTRAN 90 features such as allocatable arrays, derived types, modules and generic routines, are discussed in Sections 8.1.3–8.1.6.

- The format for internal documentation is explained in Section 8.1.7.

- Basic aspects of the model code such as the principle of key ids, time formats, data flags and the units of program variables are discussed in Section 8.2.

## 8.1 Implementation of FORTRAN 90

A main difference between versions 1 and 2 of COHERENS is that the code in the old version is written in FORTRAN 77 and the latter in FORTRAN 90. This section discusses the programming conventions based upon specific features of FORTRAN 90. Users, who have no experience with FORTRAN 90 but are familiar with the FORTRAN 77 standard may consult the many books, course notes and other publications available from commercial publishers or via the internet.

### 8.1.1 COHERENS programming conventions

1. All source code is written in "free format". This implies the following:

   - The column position is irrelevant. The adopted rule is that all program lines start at column 1, with exceptions of statements within control constructs such as IF blocks, DO loops and SELECT CASE constructs, which are intended by one TAB position to the right. Lines within a nested construct are intended with respect to the previous one. For clarity no indentation is applied if the DO/ENDDO statement of a nested loop is located just below/above the DO/ENDDO of the parent loop. This is illustrated with the following example.

```
idesc_360: DO idesc=1,MaxIOTypes
ifil_360: DO ifil=1,MaxIOFiles
   SELECT CASE (idesc)
      CASE (io_mppmod,io_modgrd,io_metgrd,io_sstgrd,io_biogrd,&
          & io_nstgrd,io_biospc,io_rlxobc,io_nstspc)
         modfiles(idesc,ifil,:)%nocoords = 0
      CASE (io_1uvsur,io_2uvobc,io_3uvobc,io_salobc,io_tmpobc,&
          & io_bioobc)
         IF (ifil.EQ.1) THEN
            modfiles(idesc,ifil,:)%nocoords = 0
         ELSE
            modfiles(idesc,ifil,:)%nocoords = 1
         ENDIF
      CASE (io_inicon,io_2uvnst,io_3uvnst,io_salnst,io_tmpnst,&
          & io_bionst,io_metsur,io_sstsur,io_biosur)
         modfiles(idesc,ifil,:)%nocoords = 1
   END SELECT
ENDDO ifil_360
ENDDO idesc_360
```

   Example 8.1: indentation of control structures and continuation lines.

   - Although the free format allows line lengths upto 132 characters, a length of 80 characters is taken for clarity since this is the maximum length on standard X-windows on UNIX/LINUX machines.

Statements longer than this limit are continued on the next line by appending a '&' character at the end of the line and at the start of the (possibly) intended line, as shown in Example 8.1.

- Comments lines start with a '!' in the first column. Although allowed by the FORTRAN 90 standard, the common practice is, for reason of clarity, not to use comments after the first column (i.e. in the middle of a line).
- Short statements may be written on one line by inserting a ';' before each next statement.
- Statements labels always start at the first column.

2. Names of variables, named constants, program units (subroutines, functions, modules) and dummy arguments can contain letters, digits and underscores. The first character must be a letter. The maximum length of a name is given by the program parameter lenname = 31, which is also the FORTRAN 90 standard. The underscore character '_' is used in the code for names composed of different words, e.g. density_equation, land_mask, or for key ids (see Section 8.2.1) below.

3. Although names in FORTRAN 90 are case insensitive, the convention is adopted to write all FORTRAN 90 specific names in uppercase, and all names, defined in the code in lowercase letters. A mixed case is used for some systems parameters (see e.g. *syspars.f90*).

4. The program uses explicit typing. This means that the type of each variable, parameter or function result needs to declared explicitly in the declaration part of a program unit. This part must be preceded by the line:

   ```
   IMPLICIT NONE
   ```

5. The indices of an array defined on the model grid, as defined in Section 10.1.1, are denoted by i,j,k for respectively the X-, Y- and Z-direction.

6. In FORTRAN 77 the values of a model grid array can only be accessed element-wise within an assignment statement. The FORTRAN 90 standard allows to use assignments on whole arrays or array sections. The rule is that the expression on the right is either a scalar or an array with the same shape as the one on the left. In the first case the value of the scalar is assigned to all elements in the array on the left. In

the second one, the values of the array expression on the right are assigned element-wise to the corrsponding elements of the array variable on the left. The convention, adopted in COHERENS, is to use array assignments where possible, e.g.

```
REAL, DIMENSION(ncloc,nrloc,nz) :: array3dc
...
array3dc = 0.0
```

This procedure is primarily used for the initialisation of variables. However, in most parts of the program distinction has to be made between grid points located on land and sea points. The method consists in maintaining a vertical ("k") loop, whereas the ("i,j") loops in the horizontal direction are replaced by array assignments within a WHERE block, as in the following example:

```
k_434: DO k=1,nz
    WHERE (maskatc_int)
      psic_A(1:ncloc,1:nrloc,k) = tridcfc(:,:,k,4)
    END WHERE
ENDDO k_434
```

Example 8.2: array assignment and land masks.

The array maskatc_int is .TRUE. or .FALSE. for grid points located at sea or on land and has the same shape as the array assignment(s) within the WHERE block. In this way calculations are restricted to sea points only.

7. Scalar and array variables must be initialised. More precisely, each scalar or array element must have a value assigned to it, before it can appear within an expression on the right of an assignment statement. The standard initialisation value for variables which have no useful default value is 0.0 for REAL, 0 for INTEGER, .FALSE. for LOGICAL and '' (empty string) for CHARACTER variables. In this way a model grid array is always defined at all grid points. Values at sea points are usually re-defined within the program whereas values in land areas remain equal to the initial (zero or .FALSE.) value.

8. As explained in Section 12.1, there are four types of program units: main program, external routines, module routines and modules where the main variables used in the program are declared. The following rules are applied in the code:

- Dummy arguments of an external or module routine need to be declared with the INTENT attribute (although this is not required by the FORTRAN 90 standard). The INTENT attribute can have the values IN, OUT and INOUT.
- Optional arguments are only allowed in module routine declarations and not in external routines. This avoids the programming of explicit interfaces. Optional arguments are positioned after all non-optional arguments.
- Argument association in a routine call is positional for non-optional arguments, whereas keyword association is used for (eventually) optional arguments.

```
CALL cf90_inquire_variable(iunit,ivar,name=f90_name(ivar),&
                        & dimids=iddims(1:ndims))
```

  where name and dimids are the names of dummy optional arguments.
- Module routines are declared in a sub-program with a USE statement of the form given in Section 8.1.5.
- The last lines of a SUBROUTINE or FUNCTION consist of a RETURN statement followed either by an empty line, one or more specification or errror code lines, followed by a END proc_name statement where proc_name is the name of the routine.
- Since all variables (scalars and arrays) are to be initialised, they must be defined with a value before they can be used as an actual argument with the INTENT(IN) or INTENT(INOUT) attribute. This is of importance, since, for example, passing an undefined value as argument in a read or MPI call may produce unexpected side effects.

9. The following FORTRAN 77 features are not allowed or not recommended in the COHERENS programming convention:

   - COMMON and INCLUDE[1] are excluded
   - GOTO statements are allowed only in exceptional cases (e.g. for error coding).

[1]INCLUDE mpif.h is the only allowed exception.

### 8.1.2 Data types

Table 8.1 lists the data types implemented in COHERENS V2.0. The COMPLEX, INTEGER(KIND=8) and (derived) TYPE do not exist or are non-standard in FORTRAN 77. The following comments are to be given:

- For the LOGICAL, INTEGER, REAL, COMPLEX types without a KIND specifier, a default KIND value is taken, as given in the last column. These defaults are standard on most (commercial and free software) compilers. If this is not the case, they can usually be enforced through compiler options.

- The INTEGER (KIND=8) type is not supported by all compilers (such as `gfortran`). In that case, the KIND value is replaced 4. This poses no problem for most program applications, since the type is only used to designate the number of seconds since a given date (see Section 8.2.2 for a further discussion).

- Each type has a corresponding id, represented by an INTEGER parameter, given in the second column of Table 8.1. The id is e.g. used in the program to identify the type of variable in a routine call. For example, the last argument in the routine call

  ```
  CALL error_alloc('depmeanatc',2,(/ncloc+2,nrloc+2/),real_type)
  ```

  informs the routine error_alloc that the variable 'depmeanatc' is of type REAL.

- The KIND value is represented by a named constant, given in column 3, whose value is given in the fourth column of Table 8.1.

- DERIVED TYPES are a new feature of FORTRAN 90 further discussed in Section 8.1.4.

- COMPLEX variables are currently only used for fast Fourier analysis (file *fft_library.f90*).

Type declaration statements have the following general FORTRAN 90 syntax:

```
 type [, att,...] ::  var_name
```

where

*type* : the data type of the variable

Table 8.1: Model data types.

| FORTRAN type | COHERENS type | KIND parameter | (assumed) size in bytes |
|---|---|---|---|
| CHARACTER | char_type | kndchar | 1 |
| LOGICAL | log_type | kndlog | 4 |
| INTEGER | int_type | kndint | 4 |
| INTEGER (KIND=8) | longint_type | kndilong | 4 or 8 (non-standard) |
| REAL | real_type | kndreal | 4 |
| REAL (KIND=8) | long_type | kndlong | 8 |
| COMPLEX | complx_type | kndcmplx | 8 |
| DERIVED | – | – | |

*att* : one or more attribute(s) of the variable, which can take the following forms

SAVE: the value of the variable is saved after the routines is exited

INTENT: used to declare dummy arguments only. Takes one of the forms: INTENT(IN) if the actual argument is defined in the calling routine and not re-defined in the called routine, INTENT(OUT) of the variable is assumed to be undefined in the calling routine and becomes defined within the called routine, INTENT(INOUT) if the actual argument is defined in the calling routine and can be re-defined in the called routine. Note that the INTENT attribute is always given in declaration statements of dummy arguments (although this is not required by the FORTRAN 90 standard).

DIMENSION: used to define the shape of an array variable. The array shape is added in parentheses.

ALLOCATABLE: to declare an allocatable array (see Section 8.1.3)

PARAMETER: a named constant whose value cannot be changed by the program

*var_name* name of the variable

It is remarked that

- CHARACTER variables have an additional attribute LEN=*len* where *len* is the length of the character string.
- Variables, sharing the same attributes are (preferentially) declared on the same line.

- Declaration and definition of DERIVED TYPE variables is discussed in Section 8.1.4. To improve transparancy all DERIVED TYPE defnitions are made in the common file *datatypes.f90*.

For example

```
CHARACTER (LEN=12) :: ctype
CHARACTER (LEN=lenname), DIMENSION(MaxProgLevels) :: procname
INTEGER, PARAMETER :: lentime = 21
INTEGER, SAVE :: iunit
REAL :: xtemp, ytemp
REAL, DIMENSION(ncloc,nrloc) :: array2dc1
REAL, SAVE, ALLOCATABLE, DIMENSION(:,:) :: array2dc2, array2dc3
```

Example 8.3: examples of type declarations.

### 8.1.3 Allocatable arrays

Allocatable arrays are arrays declared with a rank but without shape. After been allocated with the ALLOCATE statement, allocated arrays can be deallocated with a DEALLOCATE statement at any time within the program. Advantages are:

- An efficient programming of ALLOCATE and DEALLOCATE statements offers the possibility for significant memory savings and a more efficient use of internal memory. For example, the COHERENS V1 program, written in FORTRAN 77, required that all memory allocations are known at compilation, implying the consumption of unused memory.
- Array bounds can (re)assigned with non-constant values.
- Allocatable arrays can be SAVEd, even before the "ALLOCATABLE" statement is executed (see Example 8.3)
- Contrary to automatic arrays which are usually stored on the machine's stack memory, allocated arrays use internal memory. This offers a clear advantage on machines with a limited stack size.
- In parallel mode, different bounds can be defined for the same array on different process domains.

Disadvantages are:

- Arrays, that have not been allocated yet, cannot be passed as arguments to a routine call.
- Errors are difficult to debug. For example, it may occur that values are assigned to an array which is allocated with a zero size. This causes a memory fault which is often difficult to detect, since it usually causes a crash at a different location within the program.
- Although this has not been observed from the performance tests performed with COHERENS V2.0, an intensive use of allocation/deallocation may decrease the performance of the program.

A source code example of array allocation is given below.

```
REAL, ALLOCATABLE, DIMENSION(:,:) :: array2d
...
l1 = ...; l2 = ...; u2 = ...
ALLOCATE (array2d(l1:l2,u2),STAT=ierr)
CALL error_alloc('array2d',2,(/l2-l1+1,u2/),real_type)
...
IF (ALLOCATED(array2d)) DEALLOCATE (array2d)
```

Example 8.4: example of an allocate statement in COHERENS.

The error_alloc routine is called by the program after each ALLOCATE statement to check whether an allocation error occurred.

Array allocation is applied in the model as follows.

- "Global arrays", i.e. arrays which are accessible to all program units such as temperature, currents, ..., are (almost) always declared with the ALLOCATE and SAVE attribute[2]. Note that in case of a parallel mode, the shapes of model grid arrays depend on the size of the subdomain.
- Local arrays are only accessible to a program unit (SUBROUTINE or FUNCTION). They are declared either as allocatable or as automatic arrays depending on whether the CPP option -DMPI is set (see Section 3.2). Consider the following example

[2]For technical reasons a few arrays are declared with constant array dimensions.

```
SUBROUTINE ....
!---declare
#ifdef ALLOC
   REAL, SAVE, ALLOCATABLE, DIMENSION(:,:,:) :: source
#else
   REAL, DIMENSION(ncloc,nrloc,nz) :: source
#endif /*ALLOC*/
...
!---allocate
#ifdef ALLOC
   ALLOCATE (source(ncloc,nrloc,nz),STAT=errstat)
   CALL error_alloc('source',3,(/ncloc,nrloc,nz/),real_type)
#endif /*ALLOC*/
.....
!---deallocate
#ifdef ALLOC
   DEALLOCATE (source)
#endif /*ALLOC*/
END SUBROUTINE
```

Example 8.5: allocation/deallocation of local arrays.

If the ALLOC option is set, the local array source is declared as ALLOCATABLE, allocated at the beginning of the routine and deallocated before exiting the routine. Otherwise, it is declared as an automatic array. The first case has to be taken if there is no sufficient memory available, the second if the allocation/deallocation procedures have a negative impact on CPU time. The choice is obviously machine-dependent.

### 8.1.4 Derived types

DERIVED TYPE variables can be considered as aggregated structures composed of "atomic" FORTRAN data types (INTEGER, REAL, LOGICAL, CHARACTER). The aim, in COHERENS, is to store all possible information about a specific item (e.g. a file or variable) in a structured format.

Before a DERIVED TYPE can be declared, its TYPE needs to be defined. The example below, taken from the source code, shows the definition of a derived type variable for storing the attributes of a program variable.

```
TYPE :: VariableAtts
   CHARACTER (LEN=lenname) :: f90_name
   CHARACTER (LEN=lendesc) :: long_name, vector_name
   CHARACTER (LEN=lenunit) :: units
   CHARACTER (LEN=lennode) :: node
   INTEGER :: data_type, ivarid, nrank
   INTEGER, DIMENSION(4) :: shape
END TYPE VariableAtts
```

Example 8.6: TYPE definition for storing variable attributes.

A variable of this type can be defined as

```
TYPE (VariableAtts) :: varatts
```

The components of varatt are accessed using the FORTRAN 90 syntax

- varatts%f90_name: a character string with the FORTRAN name of the variable
- varatts%long_name: a string of length lendesc describing the variable
- varatts%data_type: the data type of the variable as given in the second column of Table 8.1
- varatts%nrank: rank of the variable (0 for a scalar, 1 for a vector, ...)
- ...

The DERIVED TYPE FileParams is used to store all atributes of a file

```
TYPE :: FileParams
   LOGICAL :: defined, info, opened, time_regular
   CHARACTER (LEN=1) :: form, status
   CHARACTER (LEN=leniofile) :: filename, pathname
   CHARACTER (LEN=lendesc) :: filedesc
   INTEGER :: endfile, header_type, iostat, iunit, lenrec, maxrecs, &
            & nocoords, nodim, novars, timeid, timerec, tskips, &
            & varid, zetaid
   INTEGER, DIMENSION(3) :: tlims
END TYPE FileParams
```

Example 8.7: TYPE definition for storing file attributes.

A useful property of DERIVED TYPEs is that they can de declared with the same attributes as any other data type. This means that they can be declared as scalars, arrays with a given shape, allocatable arrays with a given rank and with the SAVE or INTENT attribute. The following example illustrates how the attributes of the variables within a certain data file are defined first and then written to the file.

```
!---declare
INTEGER :: numvars
TYPE (FileParams), DIMENSION(MaxIOTypes,MaxIOFiles,2) :: modfiles
TYPE (FileParams) :: filepars
TYPE (VariableAtts), ALLOCATABLE, DIMENSION(:) :: varatts

!---define file attributes
CALL set_modfiles_atts(io_modgrd,1,2)
filepars = modfiles(io_modgrd,1,2)
numvars = filepars%novars

!---open file
CALL open_filepars(filepars)

!---define variable attributes
ALLOCATE (varatts(numvars),STAT=errstat)
CALL error_alloc_struc('varatts',1,(/numvars/),'VariableAtts')
CALL set_modvars_atts(io_modgrd,1,2,varatts,numvars)

!---write variable attributes
CALL write_atts_mod(filepars,varatts,numvars)
...
!---deallocate
DEALLOCATE (varatts)
```

Example 8.8: defining and writing variable attributes to a data file.

File formats will be discussed in Chapter 9.

The next example describes how the relative coordinates of a 2-D external grid are obtained and stored. These type of coordinates are used to perform interpolation of model data to each point of the data grid and are further discussed in Sections 10.3 and 10.4.2. The following TYPEs are defined

```
!---attributes of surface grids
TYPE :: GridParams
```

```
   INTEGER :: nhtype, n1dat, n2dat
   REAL :: delxdat, delydat, x0dat, y0dat
END TYPE GridParams
!---horizontal relative coordinates
TYPE :: HRelativeCoords
   INTEGER :: icoord, jcoord
   REAL :: xcoord, ycoord
END TYPE HRelativeCoords
```

Example 8.9: DERIVED TYPE definitions for interpolation to surface grids.

Assume that the external grid is rectangular with uniform grid spacings in either direction. The grid can then be completely defined using the attributes stored in a variable of type GridParams. In particular the attributes n1dat and n2dat are the dimensions of the data grid in the X- and Y-direction. The relative coordinates of each data point are stored in a 2-D array of type HRelativeCoords using the following procedure

```
!---declare
TYPE (GridParams) :: surfacegrid
TYPE (HRelativeCoords), SAVE, ALLOCATABLE, DIMENSION(:,:) :: &
                                                   & gridcoords
!---define the external data grid
surfacegrid%n1dat = ...; surfacegrid%n2dat = ...
...
!---allocate
n1dat = surfacegrid%n1dat; n2dat = surfacegrid%n2dat
ALLOCATE (gridcoords(n1dat,n2dat),STAT=errstat)

!---evaluate and store the relative coordinates of the data grid
idat_110: DO idat=1,n1dat
jdat_110: DO jdat=1,n2dat
   gridcoords(idat,jdat)%icoord = ...; gridcoords(idat,jdat)%jcoord = ...
   gridcoords(idat,jdat)%xcoord = ...; gridcoords(idat,jdat)%ycoord = ...
ENDDO idat_110
ENDDO jdat_110
```

Example 8.10: storing the relative coordinates of an external data grid.

### 8.1.5 Modules

Modules are program units which can be used within a FORTRAN 90 program in two ways. Firstly, variables which need to be accessible in different program routines can be declared within a module. These types are further denored as “declaration modules”. In COHERENS V1, all variables with a global scope were stored in a COMMON block located in a *.inc* file and are made accessible with a INCLUDE statement. In COHERENS V2.0 there are no COMMON blocks any more, since the declarations are made within a module and become accessible to a program unit by putting the appropriate USE statement.

As an example, the example below shows the declarations given in the module currents for all arrays related to currents

```
MODULE currents

IMPLICIT NONE

REAL, ALLOCATABLE, DIMENSION(:,:) :: p2dbcgradatu, udevint, udfvel, udvel, &
                                   & udvel_old, umpred, umvel
REAL, ALLOCATABLE, DIMENSION(:,:) :: p2dbcgradatv, vdevint, vdfvel, vdvel, &
                                   & vdvel_old, vmpred, vmvel
REAL, ALLOCATABLE, DIMENSION(:,:,:) :: p3dbcgradatu, ufvel, uvel, uvel_old
REAL, ALLOCATABLE, DIMENSION(:,:,:) :: p3dbcgradatv, vfvel, vvel, vvel_old
REAL, ALLOCATABLE, DIMENSION(:,:,:) :: wvel, wphys

SAVE

END MODULE currents
```

Example 8.11: contents of the currents module.

If one or more of these arrays are used in a program unit, a USE statement must appear in the declaration part:

```
USE currents
```

Other types of application of the FORTRAN 90 module concept are the so-called “module routines”. These routines have the same form and purpose as the usual external SUBROUTINE and FUNCTION subprograms, except that:

- Module routines accept optional arguments, keyword arguments, array valued function results and can be used to construct generic interfaces without the need to program explicit interfaces.

- Contrary to external sub-programs, module routines require that a proper USE statement must be given in the calling sub-program.

Module routines are mainly used in COHERENS to construct so-called "libraries", i.e. an ensemble of routines with a general common purpose usually implemented through generic interfaces. They are quasi-independent of the main source code. For example, all specific MPI and netCDF routine calls are located in the files *MPI_comms.F90* and *cf90_routines.F90*. In this way, only one of these files has to be re-programmed when an newer version of the MPI or netCDF is implemented in the future. A list of module routine files is given in Table 12.2.

## 8.1.6 Generic procedures

Generic procedures group a series of procedures with similar functionality together under a common name. This generalises the FORTRAN 77 concept of INTRINSIC routines. The generic name is defined via an INTERFACE statement block. This is illustrated by the following example of the read_vars generic routine

```
MODULE inout_routines
....
INTERFACE read_vars
   MODULE PROCEDURE read_vars_int_0d, read_vars_int_1d, &
                  & read_vars_int_2d, read_vars_int_3d, &
                  & read_vars_int_4d, read_vars_real_0d, &
                  & read_vars_real_1d, read_vars_real_2d, &
                  & read_vars_real_3d, read_vars_real_4d
END INTERFACE
```

Example 8.12: definition of a generic routine through an INTERCACE block statement.

The generic routine read_vars is called when data have to be read from an external data file in COHERENS standard format. The appropriate routine is selected by the program from the list in the MODULE PROCEDURE statement, depending on the type and rank of the argument which returns the data to be read. In this way the input data argument can be of type INTEGER or REAL and represent scalars or arrays of rank 1 to 4. The FORTRAN code of the specific routines with the same generic name needs to be located in the same file (*inout_routines.f90*) where the INTERFACE statement is made. Note, that although each specific routine must have the same number of non-optional arguments, the number and type of optional arguments may differ.

If a call to a generic or non-generic module routine is made, the USE statement must be given in the declaration part of the calling routine.

```
USE  module_name, ONLY:  routine_name
```

Example 8.13: syntax of a USE statement for module routines.

where *module_name* is the name of the module and *routine_name* the name of the (non)-generic routine. Consider the following example in the file *Grid_Arrays.F90*:

```
SUBROUTINE read_grid
...
USE inout_routines, ONLY: close_filepars, open_filepars, &
                        & read_glbatts_mod, read_varatts_mod, &
                        & read_vars
...
   CALL read_vars(gsigcoord,filepars,varid,varatts)
...
END SUBROUTINE read_grid
```

Example 8.14: example of a USE statement for module routines.

since gsigcoord is a REAL array of rank 3, the actually called routine is read_vars_real_3d.

### 8.1.7 Internal documentation and structured layout of the code

The COHERENS programming conventions include rules for internal documentation and code layout. These rules and some of the conventions discussed in the sections above are illustrated in Example 8.15 which shows the complete code of routine Zdif_at_C which calculates the vertical diffusion term in a scalar transport equation. A line number is added for illustrative purposes in the first four columns at each line in the example. This means that the actual code line (in this example only) starts in column 5. Note that the numbers in the discussion below refer to specific lines.

Five parts can be distinguished: a header with the routine declaration and comments, declaration part, initialisation, "main" code lines and finalisation.

**header** lines 1–19 with the following information:

- name of the routine and a short description
- name(s) of the author(s)
- a more detailed description (if needed) under the Description header

- reference to a publication or section of the user manual
- the name(s) of the sub-programs calling the routine
- the name(s) of the routines called in this routines (the routines log_timer_in and log_timer_out are called in most routines and may be omitted here)

**declaration part** This consists of the following segments

- lines 20–29: USE statements. The ONLY attribute is given for module routines only.
- lines 30–32 with the IMPLICIT NONE statement
- lines 33–44: declaration of the arguments with the INTENT attribute (preceded by the *Arguments* comment line)
- lines 45–74: comment lines with a description of the arguments in a three column format (name, type, purpose), the unit of the variable is given at the end of the line
- lines 75–88: declaration of all local variables. Except for a few exceptions in the program, local arrays are declared within a #ifdef block, either as allocatable or automatic arrays depending on the status of the -DALLOC compiler option.
- lines 89–96: internal documentation of the most meaningfull local variables using the same three column format
- lines 97–98: two blank lines to make a clear separation between the header and the program code itself

**initialisation**

- lines 99–101: write information to the log-file
- lines 102–121: allocate local arrays in case -DALLOC is defined. Note that each ALLOCATE statement is checked for errors.
- lines 122–161: initialise parameters and arrays

**main code**

- lines 162–279: calculation of the vertical diffusion term in a scalar transport equation (including boundary conditions)

**finalisation**

- lines 280–289: deallocation of local arrays if -DALLOC is defined.
- line 290: write information to the log-file

- lines 291–292: two blank lines
- line 293: the RETURN statement is not required by the FORTRAN 90 standard, but has been implemented in the COHERENS programming convention
- line 294: blank line
- line 295: END statement followed by the name of the routine (the latter is not required in FORTRAN 90 but included in the convention for clarity)

**sectioning**

The actual code, i.e. excluding the declaration part, is divided into numbered sections, subsections, subsubsections. DO loop blocks within one of these sections has a label composed of the name of the iteration counter followed by _, followed by the section number. An extra number is attached is there are more than one DO loops within a section, subsection, .... For example the k-loop which starts on line 218, is the second one in subsection 3.3 and has the label k_332.

```
SUBROUTINE Zdif_at_C(psic,tridcfc,vdifcoefatw,novars,ibcsur,ibcbot,nbcs,&
                    & nbcb,bcsur,bcbot,kbounds)
!************************************************************************
!
! *Zdif_at_C* Vertical diffusion term for a quantity at C-nodes
!
! Author - Patrick Luyten
!
! Description -
!
! Reference -
!
! Calling program - transport_at_C_3d, transport_at_C_4d1,
!                   transport_at_C_4d2
!
! Module calls - error_alloc
!
!************************************************************************
!
USE depths
USE grid
USE gridpars
USE iopars
USE physpars
USE switches
USE syspars
USE timepars
USE error_routines, ONLY: error_alloc
USE time_routines, ONLY: log_timer_in, log_timer_out

IMPLICIT NONE

!
!* Arguments
!
INTEGER, INTENT(IN) :: ibcbot, ibcsur, nbcb, nbcs, novars
INTEGER, INTENT(IN), DIMENSION(2) :: kbounds
REAL, INTENT(IN), DIMENSION(1-nhalo:ncloc+nhalo,1-nhalo:nrloc+nhalo,&
                          & nz,novars) :: psic
REAL, INTENT(INOUT), DIMENSION(ncloc,nrloc,nz,4,novars) :: tridcfc
REAL, INTENT(IN), DIMENSION(ncloc,nrloc,nz+1) :: vdifcoefatw
```

```
REAL, INTENT(IN), DIMENSION(ncloc,nrloc,nbcs,novars) :: bcsur
REAL, INTENT(IN), DIMENSION(ncloc,nrloc,nbcb,novars) :: bcbot

!
! Name        Type    Purpose
!----------------------------------------------------------------------
!*psic*       REAL    C-node quantity to be diffused [psic]
!*tridcfc*    REAL    Tridiagonal matrix for implicit vertical solution
!*vdifcoefatw*REAL    Diffusion coefficient at W-nodes [m^2/s]
!*novars*     INTEGER Number of variables
!*ibcsur*     INTEGER Type of surface boundary condition
!                 = 0 => Neumann (zero flux)
!                 = 1 => Neumann (prescibed flux)
!                 = 2 => Neumann (using transfer velocity)
!                 = 3 => Dirichlet at first C-node below the surface
!                 = 4 => Dirichlet at the surface
!*ibcbot*     INTEGER Type of bottom boundary condition
!                 = 0 => Neumann (zero flux)
!                 = 1 => Neumann (prescibed flux)
!                 = 2 => Neumann (using transfer velocity)
!                 = 3 => Dirichlet at first C-node above the bottom
!                 = 4 => Dirichlet at the bottom
!*nbcs*       INTEGER Last dimension of array bcsur
!*nbcb*       INTEGER Last dimension of array bcbot
!*bcsur*      REAL    Data for surface boundary condition
!             (:,:,1,:) => prescribed surface flux or surface value
!             (:,:,2,:) => transfer velocity [m/s]
!*bcbot*      REAL    Data for bottom boundary condition
!             (:,:,1,:) => prescribed surface flux or surface value
!             (:,:,2,:) => transfer velocity [m/s]
!*kbounds*    INTEGER Vertical bounds
!
!----------------------------------------------------------------------
!
!*Local variables
!
INTEGER :: ivar, k, kmax, kmin, npcc
REAL :: theta_vdif1, xexp, ximp
#ifdef ALLOC
   REAL, SAVE, ALLOCATABLE, DIMENSION(:,:) :: array2dc1, array2dc2, &
                                            & array2dc3
   REAL, SAVE, ALLOCATABLE, DIMENSION(:,:,:) :: array3d, difflux
#else
   REAL, DIMENSION(ncloc,nrloc) :: array2dc1, array2dc2, array2dc3
   REAL, DIMENSION(ncloc,nrloc,2:nz) :: array3d
   REAL, DIMENSION(ncloc,nrloc,nz+1) :: difflux
#endif /*ALLOC*/

!
! Name       Type  Purpose
!------------------------------------------------------------------------------
!*difflux*   REAL  Diffusive flux (times factor) at W-nodes    [m^2/psic]
!
!------------------------------------------------------------------------------
!


procname(pglev+1) = 'Zdif_at_C'
CALL log_timer_in(npcc)

!
!1. Allocate arrays
!------------------
!

#ifdef ALLOC
   ALLOCATE (array2dc1(ncloc,nrloc),STAT=errstat)
   CALL error_alloc('array2dc1',2,(/ncloc,nrloc/),real_type)
   ALLOCATE (array2dc2(ncloc,nrloc),STAT=errstat)
   CALL error_alloc('array2dc2',2,(/ncloc,nrloc/),real_type)
   ALLOCATE (array2dc3(ncloc,nrloc),STAT=errstat)
   CALL error_alloc('array2dc3',2,(/ncloc,nrloc/),real_type)
   ALLOCATE (array3d(ncloc,nrloc,2:nz),STAT=errstat)
   CALL error_alloc('array3d',3,(/ncloc,nrloc,nz-1/),real_type)
   IF (iopt_vdif_impl.NE.2) THEN
      ALLOCATE (difflux(ncloc,nrloc,nz+1),STAT=errstat)
      CALL error_alloc('difflux',3,(/ncloc,nrloc,nz+1/),real_type)
   ENDIF
#endif /*ALLOC*/

!
!2. Initialise
```

```
!-------------
!
!---fluxes
IF (iopt_vdif_impl.NE.2) THEN
   WHERE (maskatc_int)
      difflux(:,:,1) = 0.0
      difflux(:,:,nz+1) = 0.0
    END WHERE
ENDIF

!---work space arrays
WHERE (maskatc_int)
   array2dc1 = deptotatc(1:ncloc,1:nrloc)**2
END WHERE
k_201: DO k=2,nz
    WHERE (maskatc_int)
       array3d(:,:,k) = vdifcoefatw(:,:,k)&
                      & /(array2dc1*delsatw(1:ncloc,1:nrloc,k))
    END WHERE
ENDDO k_201

IF (ibcsur.LE.2) THEN
   WHERE (maskatc_int)
     array2dc2 = deptotatc(1:ncloc,1:nrloc)*delsatc(1:ncloc,1:nrloc,nz)
   END WHERE
ENDIF

IF (ibcbot.LE.2) THEN
   WHERE (maskatc_int)
      array2dc3 = deptotatc(1:ncloc,1:nrloc)*delsatc(1:ncloc,1:nrloc,1)
   END WHERE
ENDIF

!---time factors
xexp = delt3d*(1.0-theta_vdif)
ximp = delt3d*theta_vdif
theta_vdif1 = 1.0-theta_vdif

!
!3. Diffusion terms
!------------------
```

```
!

ivar_300: DO ivar=1,novars

!
!3.1 Explicit fluxes
!-------------------
!

   IF (iopt_vdif_impl.NE.2) THEN
       k_310: DO k=2,nz
           WHERE (maskatc_int)
              difflux(:,:,k) = xexp*array3d(:,:,k)*&
                            & (psic(1:ncloc,1:nrloc,k,ivar)-&
                             & psic(1:ncloc,1:nrloc,k-1,ivar))
           END WHERE
      ENDDO k_310
   ENDIF

!
!3.2 Explicit terms
!-----------------
!

   IF (iopt_vdif_impl.NE.2) THEN
      k_320: DO k=kbounds(1),kbounds(2)
          WHERE (maskatc_int)
             tridcfc(:,:,k,4,ivar) = tridcfc(:,:,k,4,ivar) + &
                                   & (difflux(:,:,k+1)-difflux(:,:,k))&
                                   & /delsatc(1:ncloc,1:nrloc,k)
          END WHERE
      ENDDO k_320
   ENDIF

!
!3.3 Implicit terms
!------------------
!

   IF (iopt_vdif_impl.NE.0) THEN

```

```
!     ---lower flux
      kmax = MERGE(nz,nz-1,ibcsur.LE.2)
      k_331: DO k=2,kmax
          WHERE (maskatc_int)
             array2dc1 = ximp*array3d(:,:,k)/delsatc(1:ncloc,1:nrloc,k)
             tridcfc(:,:,k,1,ivar) = tridcfc(:,:,k,1,ivar) - array2dc1
             tridcfc(:,:,k,2,ivar) = tridcfc(:,:,k,2,ivar) + array2dc1
          END WHERE
      ENDDO k_331

!     ---upper flux
      kmin = MERGE(1,2,ibcbot.LE.2)
      k_332: DO k=kmin,nz-1
          WHERE (maskatc_int)
             array2dc1 = ximp*array3d(:,:,k+1)&
                       & /delsatc(1:ncloc,1:nrloc,k)
             tridcfc(:,:,k,2,ivar) = tridcfc(:,:,k,2,ivar) + array2dc1
             tridcfc(:,:,k,3,ivar) = tridcfc(:,:,k,3,ivar) - array2dc1
          END WHERE
      ENDDO k_332

   ENDIF

!
!3.4 Boundary conditions
!----------------------
!
!3.4.1 Surface
!-------------
!
!  ---Neumann (prescribed flux)
   IF (ibcsur.EQ.1) THEN
      WHERE (maskatc_int)
         tridcfc(:,:,nz,4,ivar) = tridcfc(:,:,nz,4,ivar) + &
                                & delt3d*bcsur(:,:,1,ivar)/array2dc2
      END WHERE

!  ---Neumann (using transfer velocity)
   ELSEIF (ibcsur.EQ.2) THEN
      WHERE (maskatc_int)
         tridcfc(:,:,nz,2,ivar) = tridcfc(:,:,nz,2,ivar) + &
```

```
                                & ximp*bcsur(:,:,2,ivar)/array2dc2
         tridcfc(:,:,nz,4,ivar) = tridcfc(:,:,nz,4,ivar) - &
                                & delt3d*bcsur(:,:,2,ivar)*&
                          & (theta_vdif1*psic(1:ncloc,1:nrloc,nz,ivar)-&
                                & bcsur(:,:,1,ivar))/array2dc2
      END WHERE
   ENDIF

!
!3.4.2 Bottom
!------------
!
!  ---Neumann (prescribed flux)
   IF (ibcbot.EQ.1) THEN
      WHERE (maskatc_int)
         tridcfc(:,:,1,4,ivar) = tridcfc(:,:,1,4,ivar) - &
                               & delt3d*bcbot(:,:,1,ivar)/array2dc3
      END WHERE

!  ---Neumann (using transfer velocity)
   ELSEIF (ibcbot.EQ.2) THEN
      WHERE (maskatc_int)
         tridcfc(:,:,1,2,ivar) = tridcfc(:,:,1,2,ivar) + &
                               & ximp*bcbot(:,:,2,ivar)/array2dc3
         tridcfc(:,:,1,4,ivar) = tridcfc(:,:,1,4,ivar) - &
                               & delt3d*bcbot(:,:,2,ivar)*&
                          & (theta_vdif1*psic(1:ncloc,1:nrloc,1,ivar)-&
                               & bcbot(:,:,1,ivar))/array2dc3
      END WHERE
   ENDIF

ENDDO ivar_300

!
!4. Deallocate arrays
!--------------------
!

#ifdef ALLOC
   DEALLOCATE (array2dc1,array2dc2,array2dc3,array3d)
   IF (iopt_vdif_impl.NE.2) DEALLOCATE (difflux)
```

```
#endif /*ALLOC*/

CALL log_timer_out(npcc,itm_vdif)


RETURN

END SUBROUTINE Zdif_at_C
```

Example 8.15: example layout and internal documentation of a COHERENS routine.

Declaration modules have a similar layout and internal documentation, except that the code only consists of the MODULE *module_name* declaration on the first, type declarations and the END MODULE *module_name* statement on the last line. This is illustrated in Example 8.16.

```
MODULE density
!************************************************************************
!
! *density* Density arrays
!
! Author - Patrick Luyten
!
! Version - @(COHERENS)density.f90  V2.0
!
! Description -
!
!************************************************************************
!

IMPLICIT NONE

REAL, ALLOCATABLE, DIMENSION(:,:,:) :: beta_sal, beta_temp, dens, sal, temp

SAVE

!
! Name      Type  Purpose
!--------------------------------------------------------------------------
!*beta_sal* REAL  Salinity expansion coefficient                   [1/PSU]
!*beta_temp*REAL  Temperature expansion coefficient              [1/deg C]
```

```
!*dens*     REAL  Mass density                                      [kg/m^3]
!*sal*      REAL  Salinity                                             [PSU]
!*temp*     REAL  Temperature                                        [deg C]
!
!*************************************************************************

END MODULE density
```

Example 8.16: example layout and internal documentation of a COHERENS declaration module.

## 8.2 Specific program features

### 8.2.1 Key ids

Key ids are named constants which refer to a specific item, such as a variable, file class, tidal constituent or error code. The name of a key id is composed by the name of the general "class" to which the item belongs, followed by a '_' and its specific name. Some examples are given below.

1. Variable key ids have the common class name iarr. Their specific names are the same as the FORTRAN name. For example, iarr_sal is the key id of the program variable 'sal' which is salinity. Variable ids are used to define the variables used to set up user-defined output (see Chapter 20 for details).

2. Key ids referring to tidal constituents belong to the class icon_. The specific name is given by the constituent's traditional name, e.g. icon_M2 for the $M_2$ tide. Tidal key ids are a practical tool for defining the tidal constituents applied in the open boundary conditions or for selecting the constituents for the astronomical tidal force.

3. The class name for model forcing files is 'io_'. The specific name refers to the type of forcing. For example, io_metsur is the key id for the "ensemble" of properties related to the meteorological surface data file.

A list of available key classes is displayed in Table 8.2.

### 8.2.2 Date and time formats

Time can be represented in the program in four different formats. The first two are absolute calendar date and times. The next two are relative times with respect to a reference date.

Table 8.2: List of available key classes

| Class | Purpose | Defined in | Examples |
|---|---|---|---|
| io_ | attributes of a forcing file | *iopars.f90* | io_modgrd (model grid file) |
| ics_ | initial conditions | *iopars.f90* | ics_phys (physical initial conditions) |
| igrd_ | surface grids | *iopars.f90* | igrd_meteo (meteo data grid) |
| ierrno_ | error codes | *iopars.f90* | ierrno_alloc (allocation error) |
| iarr_ | model variables | *modids.f90* | iarr_temp (temperature) |
| icon_ | tidal constituents | *tide.f90* | icon_O1 ($O_1$ constituent) |
| itm_ | timers for timing report | *iopars.f90* | itm_hydro (hydrodynamics) |

1. A string format in the form of a string of lentime (23) characters: 'yyyy/mm/dd;hh:mm:ss:mmm' where yyyy = year, mm = month, dd = day in month, hh = hour in day, mm = minutes, ss = seconds, mmm = milliseconds.

   - Examples

     ```
     CHARACTER (LEN=lentime) :: CDateTime, CEndDateTime, &
                              & CStartDatetime
     ```

     where the first variable represents the current date (updated at each 2-D time step), the next two respectively the end and start date of the simulation, defined by the user, e.g.

     ```
     CStartDatetime = '2009/06/15;05:09:00:000'
     CEndDatetime = '2009/07/01;15:45:06:000'
     ```

   - These time formats are part of the model setup and are used to calculate the solar altitude for evaluation of surface solar irradiance and as date/time stamp in all time series input/output.
   - Precision is 1 millisecond.
   - The separators need to be at the correct positions, their values are unimportant.

2. A vector INTEGER format in the form of a vector with 7 elements: year, month, day, hour, minutes, seconds, milliseconds

   - Examples

     ```
     INTEGER, DIMENSION(7) :: IDateTime, IEndDateTime, &
                            & IStartDatetime
     ```

     which have the same meaning as above.

- The format is only used to perform internal date/time calculations.
- Precision is 1 millisecond.

3. A scalar INTEGER format

```
INTEGER (KIND=kndilong) :: nosecsrun
```

representing the number of seconds since the start of the simulation.

- Used internally to compare the date/time in a data file with the current one in the simulation.
- Lower precision is 1 second, upper precision ∼68 years if longint = 4 or (practically) unlimited otherwise (longint = 8).

4. A scalar INTEGER ("index") format

```
INTEGER :: nt
```

which equals the current time "index" defined as the number of (2-D) time steps since the start of the simulation.

- Used both internally as externally by the user to set output times.
- Precision is the number of seconds within one time step. This variable is defined in single precision.

The following additional time parameters are used

| | |
|---|---|
| REAL :: delt2d | User defined 2-D time step in seconds. Although defined as a REAL variable, to prevent rounding errors in the calculation of the calendar date and time its precision has been restricted to 1 millisecond for time steps smaller than 1000 seconds and to 1 second otherwise. |
| REAL :: time_zone | User defined time zone, i.e. difference of local time with respect to GMT in hours. Difference is positive (negative) eastwards (westwards) from Greenwich. Default is 0. The program assumes that the start, end and current date and time within the program are given in local time. Important to note is that the date and time, obtained from external data files, must be given with respect to the same time zone as the one used in the program. |
| INTEGER :: ic3d | User defined number of 2-D time steps within one 3-D time step. |

### 8.2.3 Data flags

Data flags are commonly used in observational data sets for representing invalid data. They have been introduced in the COHERENS code to represent undefined values. The following variables are defined in the program for the flagging of model variables

| | |
|---|---|
| REAL :: real_fill | Large negative number used for flagging of REAL model variables. |
| REAL:: real_min | Large negative number used to determine whether a real variable is flagged. More specifically, the variable X is taken as flagged if X≤real_min. Obviously, real_fill<real_min. |
| INTEGER :: int_fill | Large negative number used for flagging of INTEGER model variables. |
| LOGICAL :: log_fill | Large negative number used for flagging of LOGICAL model variables. |

Data flags are used (e.g.) for the following purposes:

- If a data value in a vertical open boundary profile has been flagged, a zero gradient condition is applied at that particular point, and the data value is no longer considered as an external value.
- Flagged values in a SST forcing file are automatically replaced by the modelled temperature at the highest (near surface) level.
- Flags are used to disable interpolation at external locations.
- Flagging of some user-defined model parameters has been implemented as a practical utility in the program. It informs the program that the parameter has some specific value unknown by the user, but known to the program.

Although most model grid arrays are not defined on land areas, no flags are, for practical reasons, applied in this case. This behaviour may be changed in future versions where land values can be flagged with the netCDF _FillValue attribute.

### 8.2.4 Variable units

Variable units are based on the ‘kg-meter-sec-PSU’ system. Table 8.3 represents the units of the principle variables used in the program. Note that the unit for shear stress has, for convenience, been normalised with the (reference) density.

Table 8.3: Units of principal variables

| variable type | unit |
|---|---|
| length | m |
| time | s |
| mass | kg |
| currents | m/s |
| transports | $m^2/s$ |
| temperature | deg C |
| salinity | PSU (=$10^{-3}$kg/kg) |
| angle | radians |
| density | $kg/m^3$ |
| horizontal coordinates | m or fractional degrees |
| vertical coordinates | m or dimensionless |
| pressure | Pa |
| diffusion coefficients | $m^2/s$ |
| frequencies | radians/s |
| acceleration | $m/s^2$ |
| stress | $m^2/s^2$ (Pa divided by density) |
| heat flux | $W/m^2$ |
| salinity flux | PSU m/s |
| turbulent energy | $m^2/s^2$ (or J/kg) |
| turbulent dissipation | $m^2/s^3$ (or W/kg) |