GRID User Manual

Astrid Kerkweg[1] **& Patrick Jöckel**[2]

[1] Institut für Physik der Atmosphäre
Johannes Gutenberg Universität Mainz
55099 Mainz, Germany
kerkweg@uni-mainz.de
[2] Deutsches Zentrum für Luft-und Raumfahrt (DLR),
Institut für Physik der Atmosphäre,
Oberpfaffenhofen, D-82234 Weßling, Germany
patrick.joeckel@dlr.de

This manual is available as electronic supplement of our article "The infrastructure MESSy submodels GRID (v1.0) and IMPORT (v1.0) " in Geosci. Model Dev. (2015), available at: http://www.geosci-model-dev.net

Date: July 17, 2015

Contents

1 Introduction

This is the User Manual of the generic MESSy submodel GRID. GRID is used by submodels requiring grid transformations. Therefore it is internally used within MESSy and a model user need not to bother about the technical details of GRID. However, code developers, implementing grid transformation or other grid relevant routines into a MESSy code, may find useful information by reading this manuscript as it provides a detailed description of the data types, the modules and subroutines of GRID and its subsubmodels. Section 2 gives an overview of how the different subroutines and modules play together. The details about all files in the submodel core layer (SMCL) and the basemodel interface layer (BMIL) are provided in Sects. 3 and 4, respectively.

2 Overview

Figure 1 depicts the dependencies of the GRID submodel. The basic GRID modules[1] (`MESSY_MAIN_GRID` and `MESSY_MAIN_GRID_NETCDF`) are used by nearly every GRID module (purple arrows). Only the subsubmodels GRID_TRAFO_SCRP_BASE is completely autonomeous.

Table 1: List of routines and type declarations in GRID. Routines are coloured blue and structures red.

# Routine name	Short description	Sect.
	MESSY_MAIN_GRID_NETCDF	3.1
t_narray	definition of data type array (`t_narray`)	3.1.1.1
t_ncdim	definition of data type dimension (`t_ncdim`)	3.1.1.2
t_ncatt	definition of data type attribute (`t_ncatt`)	3.1.1.3
t_ncvar	definition of data type variable (`t_ncvar`)	3.1.1.4
INIT_NCDIM	initialisation of a variable of type `t_ncdim`	3.1.2.1
INIT_NCATT	initialisation of a variable of type `t_ncatt`	3.1.2.1
INIT_NCVAR	initialisation of a variable of type `t_ncvar`	3.1.2.1
INIT_NARRAY	initialisation / allocation of a variable of type `t_narray`	3.1.2.2
COPY_NCDIM	copying of a variable of type `t_ncdim`	3.1.2.3
COPY_NCATT	copying of a variable of type `t_ncatt`	3.1.2.3
COPY_NCVAR	copying of a variable of type `t_ncvar`	3.1.2.3
COPY_NARRAY	copying of a variable of type `t_narray`	3.1.2.3
PRINT_NCDIM	printing of a variable of type `t_ncdim`	3.1.2.4
PRINT_NCATT	printing of a variable of type `t_ncatt`	3.1.2.4
PRINT_NCVAR	printing of a variable of type `t_ncvar`	3.1.2.4
PRINT_NARRAY	printing of a variable of type `t_narray`	3.1.2.4
QCMP_NCDIM	comparison of variables of type `t_ncdim`	3.1.2.5
QCMP_NCATT	comparison of variables of type `t_ncatt`	3.1.2.5
QCMP_NCVAR	comparison of variables of type `t_ncvar`	3.1.2.5
QCMP_NARRAY	comparison of variables of type `t_narray`	3.1.2.5
IMPORT_NCDIM	import of a variable of type `t_ncdim`	3.1.2.6.1
IMPORT_NCATT	import of a variable of type `t_ncatt`	3.1.2.6.2

[1]Following the MESSy naming convention, the Fortran files are named as the modules they contain.

Table 1: List of routines in GRID (... continued)

# Routine name	Short description	Sect.
IMPORT_NCVAR	import of a variable of type t_ncvar	3.1.2.6.3
EXPORT_NCDIM	export of a variable of type t_ncdim	3.1.2.7
EXPORT_NCATT	export of a variable of type t_ncatt	3.1.2.7
EXPORT_NCVAR	export of a variable of type t_ncvar	3.1.2.7
ADD_NCATT	definition of an attribute variable	3.1.2.8
QDEF_NCVAR	check of definition status of variable of type t_ncvar	3.1.2.9
SCAN_NCVAR	import of all variables contained in a netCDF file	3.1.2.10
RENAME_NCVAR	renaming of a variable of type t_ncvar	3.1.2.11
IDX2FRAC_NCVAR	preparation for 'IXF' and 'IDX' remapping	3.1.2.12
MAXFRAC2IDX_NCVAR	postprocessing of data required for 'IDX' remapping	3.1.2.13
EXTRACT_NCATT	conversion of GRID to CHANNEL attributes	3.1.2.14
NFERR	output of netCDF error messages	3.1.3.1
STRING	conversion of a 1D character pointer to string	3.1.3.2
SORT_NARRAY	Sorting of a 1D variable of type t_narray	3.1.3.3
REORDER_NARRAY	Reordering of a variable of type t_narray	3.1.3.4
DP_NARRAY	conversion of a variable of type t_narray to double precision float (VTYPE_DOUBLE)	3.1.3.5
SP_NARRAY	conversion of a variable of type t_narray to single precision float (VTYPE_REAL)	3.1.3.6
SCALE_NARRAY	Scaling of a variable of type t_narray	3.1.3.7
CAT_NARRAY	merging of two variables of type t_narray	3.1.3.8
POSITION	calculation of position number in 1D array	3.1.3.9
ELEMENT	calculation of the element vector of an element of given position	3.1.3.10
QSORT_I	sorting of a 1D integer data array	3.1.3.11
QSORT_B	sorting of a 1D byte data array	3.1.3.11
QSORT_R	sorting of a 1D single precision data array	3.1.3.11
QSORT_D	sorting of a 1D double precision data array	3.1.3.11
ERRMSG	error handling subroutine	3.1.3.12
RGMSG	interface of message system	3.1.3.13
MAIN_GRID_SET_MESSAGEMODE	definition of verbosity of RGMSG	3.1.3.14
NARRAYMAXVAL	determination of the maximum value of the dat component for a given variable of type t_narray	3.1.3.15
NARRAYMINVAL	determination of the minimum value of the dat component for a given variable of type t_narray	3.1.3.15
MESSY_MAIN_GRID		3.2
t_geohybgrid	definition of data type for geohybrid grids	3.2.1
INIT_GEOHYBGRID	initialisation of a variable of type t_geohybgrid	3.2.2.1
COPY_GEOHYBGRID	copying of a variable of type t_geohybgrid	3.2.2.2
PRINT_GEOHYBGRID	printing of a variable of type t_geohybgrid	3.2.2.3
IMPORT_GEOHYBGRID	import of a variable of type t_geohybgrid	3.2.2.4
EXPORT_GEOHYBGRID	export of a variable of type t_geohybgrid	3.2.2.5

Table 1: List of routines in GRID (... continued)

# Routine name	Short description	Sect.
NEW_GEOHYBGRID	definition of a new grid of type `t_geohybgrid`	3.2.2.6
COMPARE_TO_GRID	comparison of a grid of type `t_geohybgrid` to given grid components	3.2.2.7
LOCATE_GEOHYBGRID	location of grid of type `t_geohybgrid` in concatenated list of grids	3.2.2.8
CLEAN_GEOHYBGRID_LIST	deletion of the concatenated list of geohybrid grids	3.2.2.9
GRID_ERROR	error function	3.2.2.10
MESSY_MAIN_GRID_TRAFO		3.3
SWITCH_GEOHYBGRID	deletion of spacial dimensions from a grid of type `t_geohybgrid`	3.3.1
CHECK_GEOHYBGRID	consistency check for components of a grid of type `t_geohybgrid`	3.3.2
SORT_GEOHYBGRID	sorting of individual grid components from smallest to largest	3.3.3
H2PSIG	calculation of sigma or pressure levels from hybgrid pressure coordinates	3.3.4
COMPLETE_GEOHYBGRID	consistent definition of mid-points and interfaces in a grid of type `t_geohybgrid`	3.3.5
CHECK_VAR_ON_GEOHYBGRID	check if a variable of type `t_ncvar` is defined on a grid of type `t_geohybgrid`	3.3.6
SORT_GEOHYBGRID_NCVAR	sorting of a variable of type `t_ncvar` according to a sorted grid of type `t_geohybgrid`	3.3.7
PACK_GEOHYBGRID_NCVAR	packing of a variable of type `t_ncvar` according to axes and dimension information	3.3.8
BALANCE_GEOHYBGRID	copying of components of one grid of type `t_geohybgrid` to another	3.3.9
BALANCE_GEOHYBGRID_NCVAR	dimensioning of a variable of type `t_ncvar` according to a grid of type `t_geohybgrid`	3.3.10
BALANCE_GEOHYBGRID_TIME	adjustment of the time axis of two grids of type `t_geohybgrid`	3.3.11
REDUCE_INPUT_GRID	reduction of input grid of type `t_geohybgrid`	3.3.12
PHIROT2PHI	calculation of (geographical) latitude	3.3.13.2
PHI2PHIROT	calculation of rotated latitude	3.3.13.1
RLAROT2RLA	calculation of (geographical) longitude	3.3.13.3
RLAROT2RLA	calculation of rotated longitude	3.3.13.4
UVROT2UV	conversion of rotated wind vector to geographical wind vector	3.3.14
UV2UVROT	conversion of geographical wind vector to rotated wind vector	3.3.14
UVROT2UV_VEC_JLOOP	conversion of rotated wind vector to geographical wind vector operating on j-vector	3.3.14

Table 1: List of routines in GRID (... continued)

# Routine name	Short description	Sect.
MESSY_MAIN_GRID_TOOLS		3.4
RGTOOL_CONVERT	conversion of a variable of type `t_ncvar` to a 4D array	3.4.1
RGTOOL_CONVERT_DAT2VAR	conversion of a 4D array to a variable of type `t_ncvar`	3.4.2
RGTOOL_G2C	conversion of grid components to multi-dimensional arrays	3.4.3
ALINE_ARRAY	conversion of a 2D array to a 1D vector	3.4.4
DEALINE_ARRAY	conversion of a 1D vector to a 2D array	3.4.5
SET_SURFACE_PRESSURE	(re-)setting of surface pressure component in a grid of type `t_geohybgrid`	3.4.6
MESSY_MAIN_GRID_TRAFO_NRGD_BASE		3.5
MESSY_MAIN_GRID_TRAFO_NRGD		3.6
REGRID_CONTROL	driver of remapping by NREGRID	3.6.1
GEOHYBGRID_AXES	construction of axes from grid information	3.6.2
PS2PS	extraction of a variable of type `t_narray` from a variable of type `t_ncvar`	3.6.3
BALANCE_GEOHYBGRID_PS	adjustment of the surface pressure fields of two grids	3.6.4
REGRID_GEOHYBGRID_PS	mapping of the surface pressure field of one grid to another grid	3.6.5
MESSY_MAIN_GRID_TRAFO_SCRP_BASE		3.7
MESSY_MAIN_GRID_TRAFO_SCRP		3.8
t_scrip_grid	definition of a structure describing a SCRIP grid	3.8.0.1
t_scrip_data	definition of a structure combining two SCRIP grids, a mapping method and the corresponding weights	3.8.0.2
t_scrip_weights	structure containing SCRIP weights	3.8.0.3
INIT_SCRIPGRID	(re-)initialisation of a grid of type `t_scrip_grid`	3.8.1
COPY_SCRIPGRID	copying of one grid of type `t_scrip_grid` to another grid of the same type	3.8.2
DEFINE_SCRIPGRID	definition of a grid of type `t_scrip_grid`	3.8.3
CLEAN_SCRIPGRID_LIST	deletion of the concatenated list of grids of type `t_scrip_grid`	3.8.4
COMPARE_TO_SCRIPGRID	comparison of grid components to a grid of type `t_scrip_grid`	3.8.5
GEOHYB2SCRIPGRID	conversion of a grid of type `t_geohybgrid` to a grid of type `t_scrip_grid`	3.8.6
INIT_SCRIPDATA	(re-)initialisation of a structure of type `t_scrip_data`	3.8.7

Table 1: List of routines in GRID (... continued)

# Routine name	Short description	Sect.
DEFINE_SCRIPDATA	definition of a variable of type `t_scrip_data`	3.8.8
COMPARE_SCRIPDATA	comparison of individual components to a SCRIP data set of type `t_scrip_data`	3.8.9
LOCATE_SCRIPDATA	location of a SCRIP data set of type `t_scrip_data` in the concatenated SCRIP data list	3.8.10
CLEAN_SCRIPDATA_LIST	deletion of the concatenated list of data sets of type `t_scrip_data`	3.8.11
CALC_SCRIPDATA	definition of a SCRIP data set of type `t_scrip_data`	3.8.12
INIT_WEIGHTS	re-initialisation of a variable of type `t_scrip_weights`	3.8.13
CALC_SCRIP_WEIGHTS	calculation of the weights component of a SCRIP data set	3.8.14
APPLY_SCRIP_WEIGHTS	SCRIP interpolation	3.8.15
SCRIP_CONTROL	driver of remapping by SCRIP	3.8.16
INTERPOL_GEOHYBGRID_PS	interpolation of a surface pressure fields defined on one grid to another grid of type `t_scrip_grid`	3.8.17
BALANCE_CURVILINEAR_PS	adjustment of the surface pressure definition of two grids	3.8.18
CONSTRUCT_INPUT_SURF_PRESSURE	definition of a surface pressure component for a grid	3.8.19
MESSY_MAIN_GRID_MPI		3.9
MESSY_MAIN_GRID_BI		4.1
P_BCAST_GRID	broadcasting of a grid of type `t_geohybgrid`	4.1.1
MAIN_GRID_INIT_MEMORY	allocation of memory	4.1.2
MAIN_GRID_FREE_MEMORY	deallocation of memory	4.1.3
MESSY_MAIN_GRID_NETCDF_BI		4.2
P_BCAST_NCVAR	broadcasting of a variable of type `t_ncvar`	4.2.1
P_BCAST_NCATT	broadcasting of a variable of type `t_ncatt`	4.2.4
P_BCAST_NCDIM	broadcasting of a variable of type `t_ncdim`	4.2.3
P_BCAST_NARRAY	broadcasting of a variable of type `t_narray`	4.2.4

3 The SMCL GRID files

In the hierarchy of the GRID SMCL modules `MESSY_MAIN_GRID_NETCDF` is the most basic one. It provides the netCDF related type declarations and corresponding routines. The structures are used as components of more complex structures. They build the basis for all other process routines. Additionally, `MESSY_MAIN_GRID_NETCDF` contains the error output routines.

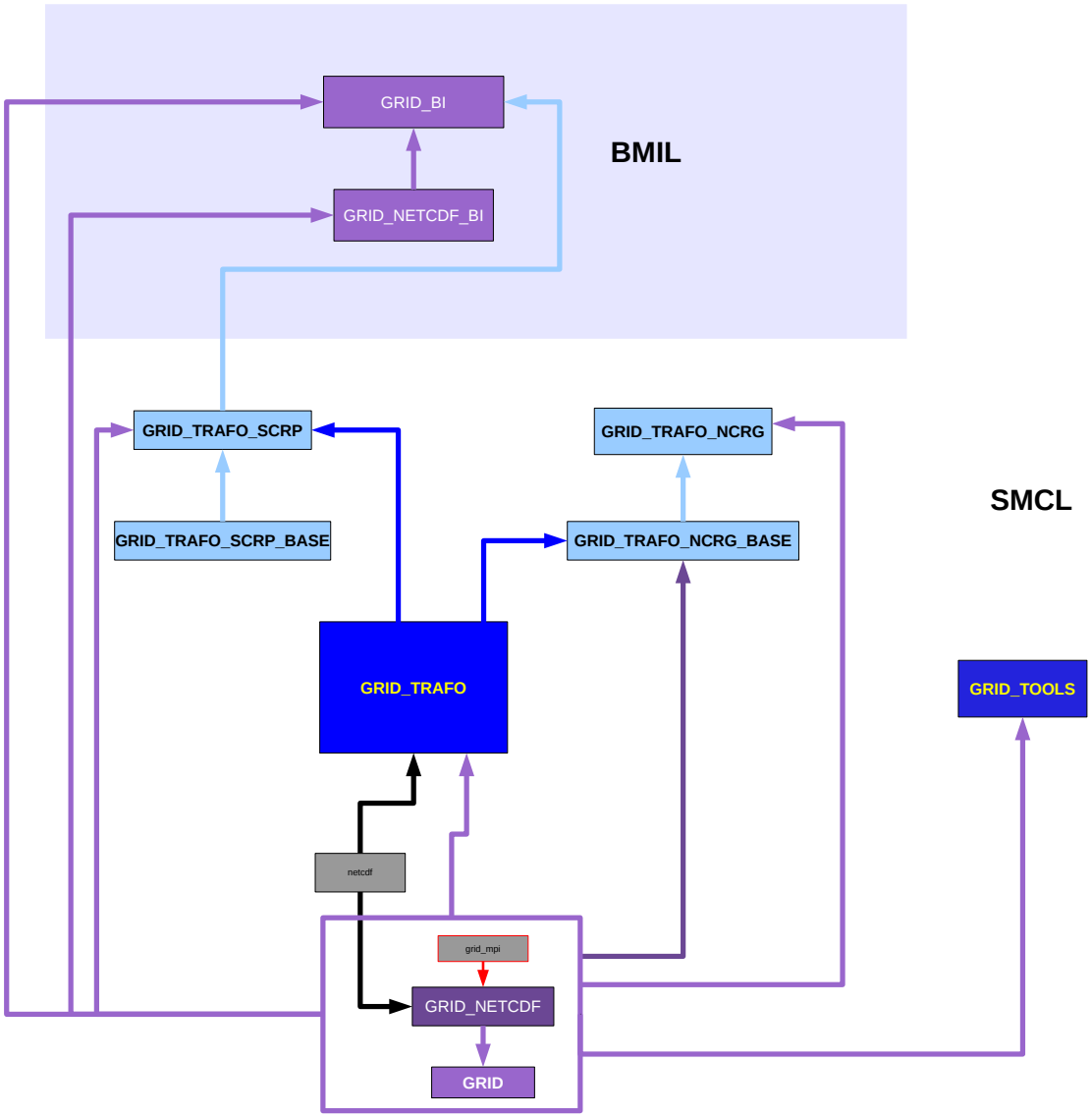

Figure 1: Diagram of dependencies of the MESSy GRID modules. The arrows point in the direction of USEage.

The module `MESSY_MAIN_GRID` contains the definition and the handling routines of the geohybrid grid. For this, the structures declared in `MESSY_MAIN_GRID_NETCDF` are used. The geohybrid grid definition and respective handling routines are the basis for the grid transformation.

The actual grid transformation routines are split into seven modules:

- Each individual mapping algorithm (at the time being SCRIP and NREGRID) consists of two module files:
  - a "`base`" file, containing the code of the actual regridding algorithm and
  - the interface file that calls those core routines and contains the subroutines required for

Parameter name	pointer name in t_narray	type / meaning
VTYPE_UNDEF		undefined, i.e. none of the pointers in the variable of type t_narray is associated
VTYPE_INT	vi	the integer pointer is associated
VTYPE_REAL	vr	the single precision pointer is associated
VTYPE_DOUBLE	vd	the double precision pointer is associated
VTYPE_BYTE	vb	the byte value (integer I4) pointer is associated
VTYPE_CHAR	vc	the character pointer is associated

Table 2: Table of variable types available in variables of the type t_narray

the pre- and postprocessing of the input and output to/from the core, according to the requirements of the respective regridding algorithm.

- In addition to the mapping algorithm dependent modules, one module (MESSY_MAIN_GRID_TRAFO) provides tools for data operations which are independent of the mapping method.
- The module MESSY_MAIN_GRID_TOOLS contains tools for conversions between the internally used 1D storage format and the user defined 2D and 4D arrays, which are attributed to geo-spacial grids of the respective model. Furthermore, a subroutine is provided for updating the surface pressure field contained in the grid definition during the simulation.
- Last but not least, the module MESSY_MAIN_GRID_MPI comprises routines for the proper abortion of the simulations, e.g., in case of error, in parallel mode.

3.1 MESSY_MAIN_GRID_NETCDF

3.1.1 GRID data types

The layout of the basic GRID data types is based on the netCDF file format definition. Thus structures for data arrays (t_narray), dimensions (t_ncdim), attributes (t_ncatt) and variables (t_ncvar) are defined here.

3.1.1.1 TYPE t_narray

The type t_narray defines a universal data array. Universal means here, that each data type can be used. For this, the type definition contains a vector pointer for each basic data type: single precision float (vr), double precision float (vd), integer(I8) (vi), integer(I4) or byte (vb), and characters (vc).

```
!------------------------------------------------------------------------------
  TYPE t_narray
     ! n-dimenional array as 1D (LINEAR) array (REAL)
     INTEGER                              :: type = VTYPE_UNDEF
     INTEGER                              :: n = 0         ! number of dimensions
     INTEGER     , DIMENSION(:), POINTER  :: dim => NULL() ! dim. vector
     REAL (SP)   , DIMENSION(:), POINTER  :: vr  => NULL() ! real values
```

```
    REAL (DP)   , DIMENSION(:), POINTER  :: vd  => NULL() ! double values
    INTEGER (I8), DIMENSION(:), POINTER  :: vi  => NULL() ! integer values
    INTEGER (I4), DIMENSION(:), POINTER  :: vb  => NULL() ! byte values
    CHARACTER,    DIMENSION(:), POINTER  :: vc  => NULL() ! char. values
  END TYPE t_narray
!------------------------------------------------------------------------------
```

Only one of the pointers will become associated, when a variable of type `t_narray` is defined. All other pointers are nullified. The integer component `type` indicates which of the data pointers is associated (see Table 2 for a list of the possible values of `type`), and the variable `n` contains the length of the data array. The integer parameter `VTPYE_UNDEF` is zero and indicates, if currently no data is associated. The component `dim` is a vector which is allocated to the number of ranks of the corresponding user variable. Each entry of `dim` contains the length of that respective dimension.

3.1.1.2 t_ncdim

The structure `t_ncdim` contains information to describe a dimension as in the netCDF standard.

```
!------------------------------------------------------------------------------
  TYPE t_ncdim
    CHARACTER(LEN=GRD_MAXSTRLEN) :: name = ''    ! name of dimension
    INTEGER                 :: id    = NULL_DIMID ! dimension ID
    INTEGER                 :: len   = 0          ! length of dimension
    LOGICAL                 :: fuid  = .false.    ! flag for "UNLIMITED" dimension
    INTEGER                 :: varid = NULL_VARID ! variable ID, if coordinate var
  END TYPE t_ncdim
!------------------------------------------------------------------------------
```

This structure consists of

- a (unique) `name`,
- an ID (`id`) indicating the dimension distinctively,
- the length of the dimension (`len`),
- a logical flag (`fiud`) indicating, if this dimension is the "unlimited" dimension,
- the variable ID (`varid`) of the corresponding dimension variable in a netCDF file, if the dimension is read from / written to a netCDF file.

The integer parameters `NULL_DIMID` and `NULL_VARID` are both "-1" and indicate if the IDs are not yet set.

3.1.1.3 t_ncatt

The structure `t_ncatt` serves to store the information for the definiton of an attribute.

```
!------------------------------------------------------------------------------
  TYPE t_ncatt
     CHARACTER(LEN=GRD_MAXSTRLEN) :: name  = ''         ! attribute name
     INTEGER                      :: ID    = 0          ! ID of attribute
     INTEGER                      :: xtype = NULL_XTYPE ! type of attribute
     INTEGER                      :: varid = NULL_VARID ! variable ID or
                                                        ! ... NF90_GLOBAL
     INTEGER                      :: len   = 0          ! length of attribute
     TYPE(t_narray)               :: dat                ! content of attribute
  END TYPE t_ncatt
!------------------------------------------------------------------------------
```

It consists of

- a (unique) `name`,
- the ID of the attribute `ID`, identifying the attribute unambiguously.
- the type of the attribute (`xtype`). If the attribute is undefined, `xtype` is set to `NULL_XTYPE`, otherwise `xtype` indicates of which type the component `dat` (as structure of type `t_narray`, see Sect. 3.1.1.1) is. Table 2 lists the possible values.
- the component `varid` is set to the netCDF ID of the respective attribute, if the attribute is read from a netCDF file. `varid = NULL_VARID` indicates that the attribute is not defined in the file, or that it has not yet been read.
- the length of the attribute variable (`len`) and
- the attribute contents (`dat`) of type `t_narray`.

3.1.1.4 t_ncvar

The structure `t_ncvar` stores the definition of a netCDF variable.

```
!------------------------------------------------------------------------------
TYPE t_ncvar
   CHARACTER(LEN=GRD_MAXSTRLEN)         :: name  = ''         ! variable name
   INTEGER                              :: id    = NULL_VARID ! variable ID
   INTEGER                              :: xtype = NULL_XTYPE ! type of variable
   INTEGER                              :: ndims = 0          ! number of dimensions
   TYPE(t_ncdim), DIMENSION(:), POINTER :: dim => NULL()      ! netCDF dimensions
   INTEGER                              :: uid   = NULL_DIMID ! unlimited dim ID
   INTEGER                              :: ustep = 0          ! step along unlim. ID
   INTEGER                              :: natts = 0          ! number of attributes
   TYPE(t_ncatt), DIMENSION(:), POINTER :: att => NULL()      ! list of attributes
   TYPE(t_narray)                       :: dat                ! content of variable
END TYPE t_ncvar
!------------------------------------------------------------------------------
```

It contains the components:

- a name (`name`), (the maximal length of the string is 200 characters,)
- the variable ID (`id`), unambiguously identifying the variable,
- an indicator of the variable type (`xtype`). Table 2 lists the possible values of xtype. If the variable is undefined, `xtype` is set to `NULL_XTYPE`.
- the number of dimensions (`ndims`),
- a 1D pointer (`dim`) of type `t_ncdim` for the definition of the dimension,
- the dimenion id of the unlimited dimension (`uid`, usually the time axis),
- the hyperslice along the unlimited axis (`ustep`),
- the number of attributes (`natts`),
- a 1D pointer (`att`) of type `t_ncatt` for the definition of the attributes,
- the variable contents (`dat`) of type `t_narray`.

3.1.2 Functions and Subroutines for type handling

3.1.2.1 INIT_NCDIM, INIT_NCATT, INIT_NCVAR

These subroutines initialise the variables of the corresponding type, i.e., they are reset to their initial default contents:

- numbers are set to zero or their "undef"-value (e.g., `xtype = NULL_XTYPE`), respectively.
- allocated memory is released and
- pointers are nullified.

3.1.2.2 INIT_NARRAY

In contrast to the previous subroutines, the components of a variable of type `t_narray` are not only deallocated and set to their default value in INIT_NARRAY, but depending on the optional parameters of the variable can be allocated to a requested data size.

```
! ------------------------------------------------------------------------------
SUBROUTINE INIT_NARRAY(na, n, dim, qtype)

  ! initialise array of type narray
  ! INPUT:
  !   - n :   number of dimensions
  !   - dim:  length of dimensions
  !   - type: type of array (REAL, REAL(dp), INTEGER, CHAR, BYTE)

  IMPLICIT NONE

  ! I/O
  TYPE (t_narray),       INTENT(INOUT)          :: na
```

```
  INTEGER,                INTENT(IN),   OPTIONAL :: n
  INTEGER, DIMENSION(:), INTENT(IN),   OPTIONAL :: dim
  INTEGER,                              OPTIONAL :: qtype
! ----------------------------------------------------------------------------
```

- `n` is the number of dimensions (i.e. `na%n`)
- The 1D integer vector `dim` provides the length of the individual dimensions of a variable. Example: for a 3D array of a sides $X \times Y \times Z$ with length of $3 \times 4 \times 2$ cells, `dim = (3,4,2)` and `na%n = 3`. The required length of a data array is $3 \times 4 \times 2 = 24$.
- If `qtype` is present and a dimension `dim` is provided, the data pointer of the corresponding type is allocated to the length as determined by the product of the dimensions.

3.1.2.3 COPY_NARRAY, COPY_NCDIM, COPY_NCATT, COPY_NCVAR

These subroutines copy the content of one variable of the corresponding data type to another variable of the same type. To avoid memory leaks, the destination variable always has the `INTENT(INOUT)` attribute and is re-initialised at the beginning of the subroutines.

3.1.2.4 PRINT_NARRAY, PRINT_NCDIM, PRINT_NCATT, PRINT_NCVAR

These subroutines provide log-file output of all components of the corresponding type. These subroutines are usually not used during a production simulation. Nevertheless, they are very helpful tools during the implementation phase.

3.1.2.5 QCMP_NARRAY, QCMP_NCDIM, QCMP_NCATT, QCMP_NCVAR

These logical functions compare two variables of the corresponding data type to each other. If all components are equal, the variables are equal and the functions return `.TRUE.`. Otherwise the functions return `.FALSE.`.

3.1.2.6 IMPORT_NCDIM, IMPORT_NCATT, IMPORT_NCVAR

These subroutines import the corresponding types from a netCDF file.

3.1.2.6.1 IMPORT_NCDIM

Parameters to the routine are a dimension variable to store the read informationen (`dim`), and, optionally, the dimension name or the dimension ID, and the name or the ID of the netCDF file. For both, the dimension and the netCDF file, either the name or the ID must be given, otherwise the subroutine produces an error messages and terminates the simulation.

```
! ----------------------------------------------------------------------------
SUBROUTINE IMPORT_NCDIM(dim, dimname, dimid, file, ncid)

  ! read structure of type t_ncdim from netcdf file

  IMPLICIT NONE

  ! I/O
```

```
  TYPE (t_ncdim),   INTENT(OUT)           :: dim      ! dimension
  CHARACTER(LEN=*), INTENT(IN),  OPTIONAL :: dimname  ! name of dimension
  INTEGER,          INTENT(IN),  OPTIONAL :: dimid    ! dimension ID
  INTEGER,          INTENT(IN),  OPTIONAL :: ncid     ! netCDF file ID
  CHARACTER(LEN=*), INTENT(IN),  OPTIONAL :: file     ! filename
! ------------------------------------------------------------------------------
```

3.1.2.6.2 IMPORT_NCATT

Parameter to the subroutine are an attribute variable to store the read information (`att`) of type `t_ncatt`. The name (`varname`) or the ID (`varid`) of the corresponding netCDF variable, the attribute name (`attname`) or the ID (`attID`), the netCDF file name (`file`) or ID (`ncid`) and a logical flag indicating, if the simulation should be terminated or continued, if the attribute is not available in the netCDF file.

```
! ------------------------------------------------------------------------------
SUBROUTINE IMPORT_NCATT(att, varname, varid, attname, attID &
                        ,file, ncid                        &
                        ,lnostop)

  ! read structure of type t_ncatt from netcdf file

  IMPLICIT NONE

  ! I/O
  TYPE (t_ncatt),   INTENT(OUT)           :: att      ! attribute
  CHARACTER(LEN=*), INTENT(IN),  OPTIONAL :: varname  ! variable name
  INTEGER,          INTENT(IN),  OPTIONAL :: varid    ! netCDF variable ID
  CHARACTER(LEN=*), INTENT(IN),  OPTIONAL :: attname  ! attribute name
  INTEGER,          INTENT(IN),  OPTIONAL :: attID    ! attribute ID
  INTEGER,          INTENT(IN),  OPTIONAL :: ncid     ! netCDF file ID
  CHARACTER(LEN=*), INTENT(IN),  OPTIONAL :: file     ! filename
  LOGICAL,          INTENT(IN),  OPTIONAL :: lnostop  ! do not stop if
                                                      ! att. does not exist
! ------------------------------------------------------------------------------
```

3.1.2.6.3 IMPORT_NCVAR

`IMPORT_NCVAR` imports a variable of type `t_ncvar` from a netCDF-file.

```
! ------------------------------------------------------------------------------
SUBROUTINE IMPORT_NCVAR(var, ustep, varname, varid, file, ncid, setuid &
                        , pstart, pcount)

  ! read variable structure of type t_ncvar from netcdf file

  IMPLICIT NONE

  ! I/O
```

```
  TYPE (t_ncvar),   INTENT(INOUT)           :: var      ! variable
  INTEGER,          INTENT(IN),  OPTIONAL   :: ustep    ! step along unlim. DIM
  CHARACTER(LEN=*), INTENT(IN),  OPTIONAL   :: varname  ! variable name
  INTEGER,          INTENT(IN),  OPTIONAL   :: varid    ! variable ID
  INTEGER,          INTENT(IN),  OPTIONAL   :: ncid     ! netCDF file ID
  CHARACTER(LEN=*), INTENT(IN),  OPTIONAL   :: file     ! filename
  INTEGER,          INTENT(IN),  OPTIONAL   :: setuid   ! set this as unlim. ID
  INTEGER, DIMENSION(2), INTENT(IN), OPTIONAL :: pstart
  INTEGER, DIMENSION(2), INTENT(IN), OPTIONAL :: pcount
! ------------------------------------------------------------------------------
```

Parameters to this subroutine are

- the variable of type `t_ncvar` to store the imported data,
- the "time step" to be read (`ustep`), i.e., the hyperslice of the unlimited dimension,
- the name (`varname`) or the ID (`varid`) of the corresponding netCDF variable,
- the netCDF file name (`file`) or ID (`ncid`),
- an integer (`setuid`), which indicates that the dimension should be treated as unlimited dimension.
- two 2D integers (`pstart` and `pcount`) defining the hyperslice of data to be read. `pstart` indicates the position at which the reading should start, `pcount` how many data points should be read from `pstart` on. This is, for instance, used to reduce the size of the input grid and thus the required memory. This is of special interest, if GRID is used in parallel domain decomposition.

3.1.2.7 EXPORT_NCDIM, EXPORT_NCATT, EXPORT_NCVAR

These subroutines export the variables of the corresponding type to a netCDF file. Parameters are the corresponding variable and the name or ID of the netCDF file. For EXPORT_NCATT, additionally, the logical parameter `clobber` can be defined, if `.TRUE.` an existing attribute will be overwritten.

3.1.2.8 ADD_NCATT

This subroutine inserts an attribute to the attribute list of a variable of type `t_ncvar`.

```
! ------------------------------------------------------------------------------
SUBROUTINE ADD_NCATT(var, name, replace, vs, vr, vd, vi, vb)

  ! add/replace attribute of name 'name' to variable of type t_ncvar

  IMPLICIT NONE

  ! I/O
  TYPE (t_ncvar),            INTENT(INOUT)          :: var
  CHARACTER(LEN=*),          INTENT(IN)             :: name
  LOGICAL,                   INTENT(IN),   OPTIONAL :: replace
  CHARACTER(LEN=*),          INTENT(IN),   OPTIONAL :: vs
```

```
  REAL   (SP), DIMENSION(:), INTENT(IN),   OPTIONAL :: vr
  REAL   (DP), DIMENSION(:), INTENT(IN),   OPTIONAL :: vd
  INTEGER(I8), DIMENSION(:), INTENT(IN),   OPTIONAL :: vi
  INTEGER(I4), DIMENSION(:), INTENT(IN),   OPTIONAL :: vb
! ------------------------------------------------------------------------
```

Parameters of the subroutine are

- the variable `var` of type `t_ncvar` to which the attribute will be added,
- the `name` of the attribute,
- an optional argument (`replace`), indicating whether an existing attribute of the same name shall be replaced by the new one,
- 1D pointers (optional) corresponding to the type of the attribute to be stored in a variable of type `t_narray` containing the actual data of the attribute.

3.1.2.9 QDEF_NCVAR

This logical function tests if a variable `var` of type `t_narray` is defined. Actually, it is tested whether the structure component `type` is equal to `VTYPE_UNDEF`:

```
  QDEF_NCVAR = (var%dat%type /= VTYPE_UNDEF)
```

3.1.2.10 SCAN_NCVAR

This subroutine reads in all variables contained in a netCDF file of file ID `ncid` or name `file`.

```
! ------------------------------------------------------------------------
SUBROUTINE SCAN_NCVAR(var, file, ncid)

  IMPLICIT NONE

  ! I/O
  TYPE (t_ncvar), DIMENSION(:), POINTER              :: var   ! variables
  CHARACTER(LEN=*),             INTENT(IN), OPTIONAL :: file  ! filename
  INTEGER,                      INTENT(IN), OPTIONAL :: ncid  ! netCDF ID
! ------------------------------------------------------------------------
```

Arguments of the subroutine are a 1D pointer `var` of type `t_ncvar` and the file name or file ID, respectively. The pointer is allocated to the number of variables found in the file. Subsequently, all variables are imported from the file.

3.1.2.11 RENAME_NCVAR

This subroutine changes the name of a variable (`var`) of type `t_ncvar` to a griven new name (`newname`).

```
! ----------------------------------------------------------------------------
SUBROUTINE RENAME_NCVAR(var, newname)

  ! rename variable of type t_ncvar

  IMPLICIT NONE

  ! I/O
  TYPE (t_ncvar),   INTENT(INOUT) :: var
  CHARACTER(LEN=*), INTENT(IN)    :: newname
! ----------------------------------------------------------------------------
```

3.1.2.12 IDX2FRAC_NCVAR

This subroutine is called before the remapping, if `'IDX'` or `'IXF'` mapping is required. This is used for data represented by integer bins or categories, e.g. soil classes.

```
! ----------------------------------------------------------------------------
SUBROUTINE IDX2FRAC_NCVAR(vi, vf, vtype)

  IMPLICIT NONE

  ! I/O
  TYPE (t_ncvar), INTENT(IN)           :: vi   ! variable with index field
  TYPE (t_ncvar), INTENT(INOUT)        :: vf   ! variable with index fraction
  INTEGER,        INTENT(IN), OPTIONAL :: vtype
! ----------------------------------------------------------------------------
```

Within the subroutine `IDX2FRAC_NCVAR` a new variable (`vf`) is defined which has one additional dimension in comparison to the input variable (`vi`). The length of the additional dimension is determined by the number of categories, e.g. the number of available soil classes. Subsequently, the new variable is initialised with zero and afterwards filled with "1" for the actual class of the input file. For example, if a soil class input field indicates that grid point `(i,j)` belongs to soil class 5, the entry in the new variable associated to the indices `(i,j,5)` is set to 1, while the entries for the other soil classes or bins (`(i,j,1)`, `(i,j,2)`, `(i,j,3)`, ...) are zero.

Remapping of such a field yields automatically to the information, which fraction of the target field is covered by a certain category (e.g. soil class).

3.1.2.13 MAXFRAC2IDX_NCVAR

This subroutine is called after the remapping, if `'IDX'` regridding is requested.

```
! ----------------------------------------------------------------------------
SUBROUTINE MAXFRAC2IDX_NCVAR(vf, vi, vtype)
```

```
  IMPLICIT NONE

  ! I/O
  TYPE (t_ncvar), INTENT(IN)         :: vf    ! index fractions
  TYPE (t_ncvar), INTENT(INOUT)      :: vi    ! index
  INTEGER,      INTENT(IN), OPTIONAL :: vtype
!
  ----------------------------------------------------------------------------
```

In this case, the information provided by the remapping of the field, created by the subroutine `IDX2FRAC_NCVAR` before the mapping, needs to be reduced, as `'IDX'` regridding requests only the category of the largest fraction, i.e. the most abundant category. For example: a soil-vegetation-scheme of an ESM can only work with one soil type per grid point. In this case only the soil class which is dominating in the overlapping grid cells of the input field is used.

The output variable `vi` is defined with one dimension less than the input variable `vf`, as the dimension for the individual categories is not required. Afterwards, the vector containing all categories for one grid point is extracted and sorted. In the end, the category with the largest fraction is assigned to the output variable `vi`.

3.1.2.14 EXTRACT_NCATT

This subroutine decomposes a variable of type `t_ncvar` into its components. This is especially used in `MESSY_MAIN_IMPORT_GRID_TOOLS_BI` to convert GRID attributes to CHANNEL attributes.

```
! ----------------------------------------------------------------------------
SUBROUTINE EXTRACT_NCATT(att, type, name, i, c, r)

  IMPLICIT NONE
  INTRINSIC :: INT, MIN, REAL, LEN

  ! I/O
  TYPE (t_ncatt),   INTENT(IN)    :: att
  INTEGER,          INTENT(OUT)   :: type ! 1: integer, 2: string, 3: real(dp)
  CHARACTER(LEN=*), INTENT(OUT)   :: name
  INTEGER,          INTENT(INOUT) :: i
  CHARACTER(LEN=*), INTENT(INOUT) :: c
  REAL(DP),         INTENT(INOUT) :: r
! ----------------------------------------------------------------------------
```

Input is the attribute of type `t_ncatt`, output are the `type` of the attribute, its `name` and —depending on the `type`— the respective integer, character or real value.

3.1.3 Functions and Subroutine for message handling

3.1.3.1 NFERR

This subroutine provides output of netCDF error messages.

```
! ------------------------------------------------------------------------------
SUBROUTINE NFERR(routine, status, no)

  ! improved netcdf error output routines

  IMPLICIT NONE

  ! I/O
  CHARACTER(LEN=*), INTENT(IN)           :: routine
  INTEGER,          INTENT(IN)           :: status
  INTEGER,          INTENT(IN), OPTIONAL  :: no
! ------------------------------------------------------------------------------
```

Input are a `status` flag, the name of the calling `routine` and an optional integer parameter, which can be used to indicate the position in the calling sequence.

3.1.3.2 STRING

This function converts a 1D character pointer `c` into a `string`. This is useful for the conversion of the 1D character pointer such as the component in the type `t_narray`.

3.1.3.3 SORT_NARRAY

This subroutine sorts the entries of a 1D variable of type `t_narray`.

```
! ------------------------------------------------------------------------------
SUBROUTINE SORT_NARRAY(na, nx, reverse)

  IMPLICIT NONE

  ! I/O
  TYPE (t_narray), INTENT(INOUT)          :: na
  TYPE (t_narray), INTENT(INOUT)          :: nx
  LOGICAL        , INTENT(IN)   ,OPTIONAL :: reverse
! ------------------------------------------------------------------------------
```

The subroutine has two modi operandi:

1. If the LOGICAL `reverse` is present and `.TRUE.`: the input variable `na` of type `t_narray` is sorted according to the indices provided by the second input variable `nx` of type `t_narray`, which basic data type needs to be integer.

2. If `reverse` is not present or `.FALSE.`, the input variable `na` of type `t_narray` is sorted from smallest to largest numbers. The output variable `nx` then contains a vector of the same length of `na` storing the original indices of the position. This index array can be used as input to `SORT_NARRAY` to sort the array into its original order (e.g. after the remapping). Additionally, it can be used as input to the subroutine `REORDER_NARRAY` to sort an additional array in the same way.

3.1.3.4 REORDER_NARRAY

This subroutine reorders the input variable `na` of type `t_narray` in a way determined by the second input variable `nx` of type `t_narray`.

```
! ------------------------------------------------------------------------------
SUBROUTINE REORDER_NARRAY(na, nx)

  IMPLICIT NONE

  ! I/O
  TYPE (t_narray), INTENT(INOUT) :: na   ! n-array to reorder
  TYPE (t_narray), INTENT(IN)    :: nx   ! index n-array
! ------------------------------------------------------------------------------
```

As `nx` provides the required indices, it has to be of type `VTYPE_INT`. Additionally, the two variables `na` and `nx` need to have the same length.

3.1.3.5 DOUBLE_NARRAY

This subroutine converts a variable of type `t_narray` to double precision float (`VTYPE_DOUBLE`). This is only possible for variables of type `VTYPE_DOUBLE`, `VTYPE_REAL`, `VTYPE_INT` and `VTYPE_BYTE`.

3.1.3.6 REAL_SP_NARRAY

This subroutine converts a variable of type `t_narray` to single precision float (`VTYPE_REAL`). This is only possible for variables of type `VTYPE_DOUBLE`, `VTYPE_REAL`, `VTYPE_INT` and `VTYPE_BYTE`.

3.1.3.7 SCALE_NARRAY

This subroutine scales a variable of type `t_narray` by a given scaling factor `sc`.

```
! ------------------------------------------------------------------------------
SUBROUTINE SCALE_NARRAY(na, sc)

  ! scale data of the variable of type t_narray by sc

  IMPLICIT NONE

  ! I/O
  TYPE (t_narray), INTENT(INOUT) :: na  ! N-array
  REAL           , INTENT(IN)    :: sc  ! scaling factor
! ------------------------------------------------------------------------------
```

This is only possible for variables of type `VTYPE_DOUBLE`, `VTYPE_REAL`, `VTYPE_INT` and `VTYPE_BYTE`. Variables of type `VTYPE_INT` and `VTYPE_BYTE` are converted to `VTYPE_REAL`.

3.1.3.8 CAT_NARRAY

This subroutine appends a variable `nb` of type `t_narray` to the variable `na` of type `t_narray`.

```
! ------------------------------------------------------------------------------
SUBROUTINE CAT_NARRAY(na, nb)

  IMPLICIT NONE

  ! I/O
  TYPE(t_narray), INTENT(INOUT) :: na
  TYPE(t_narray), INTENT(in)    :: nb
! ------------------------------------------------------------------------------
```

Therefore, `na` and `nb` have to be of the same vasic type and `nb` has to be defined. If `na` is undefined `nb` is copied to `na`. If `na` and `nb` exist and are of the same type, `nb` is appended to `na`.

3.1.3.9 POSITION

This function calculates the position number in a 1D (linear) array, given, that the array is interpreted as an n-dimensional array with dimensions of length `dim = (d1, d2, d3, ..., dN)` of the element vector `vec = (v1, v2, v3, ..., vN)`. Example: In a 1D array of dimensions $4 \times 3 \times 2$, the position of the element with element vector (2,2,1) is `6`.

```
! ------------------------------------------------------------------------------
INTEGER FUNCTION POSITION(vdim, vec)

  IMPLICIT NONE

  ! I/O
  INTEGER, DIMENSION(:), INTENT(IN) :: vdim
  INTEGER, DIMENSION(:), INTENT(IN) :: vec
! ------------------------------------------------------------------------------
```

Input to the function are the integer vector `vdim` containing the lengths of the individual dimensions of the n-dimensional array, and the vector `vec` containing the indices of the required element in the n-dimensional space.

3.1.3.10 ELEMENT

This subroutine is the reverse of the function `POSITION`. It calculates the element vector `vec` of the element with a position `n` in a 1D (linear) array, given that the array is interpreted as an n-dimensional array with dimensions of length `dim`.

```
! ------------------------------------------------------------------------------
SUBROUTINE ELEMENT(dim, n, vec)

  IMPLICIT NONE
```

```
  ! I/O
  INTEGER, DIMENSION(:), INTENT(IN)  :: dim   ! dimension vector
  INTEGER,               INTENT(IN)  :: n     ! element in linear array
  INTEGER, DIMENSION(:), POINTER     :: vec   ! element vector
! ------------------------------------------------------------------------------
```

Example: for a $4 \times 3 \times 2$ space, position `n=6` corresponds to the element vector `vec=(2,2,1)`.

3.1.3.11 QSORT_I, QSORT_B, QSORT_R, QSORT_D

These subroutines sort a given 1D data array of type integer, byte, real or double precision, respectively, from smallest to largest numbers.

```
! ------------------------------------------------------------------------------
RECURSIVE SUBROUTINE QSORT_X(data,idx,ileft,iright)

  IMPLICIT NONE

  ! I/O
  REAL (SP),    DIMENSION(:), INTENT(INOUT)        :: data   ! data to sort
  INTEGER (I8), DIMENSION(:), POINTER              :: idx    ! index list
  INTEGER (I8),               INTENT(IN), OPTIONAL :: ileft, iright
! ------------------------------------------------------------------------------
```

The subroutines `QSORT_X` (X is one of I, B, R or D) are recursive subroutines which efficiently sort a given 1D data array. The pointer `idx` contains the original index of the data point. The optional arguments `ileft` and `iright` allow for an application of the subroutine to a hyperslice of the data array, thus making the sorting algorithm more efficient.

3.1.3.12 ERRMSG

This is an error handling subroutine for the output of meaningful error messages to the log-file. applications.

```
! ------------------------------------------------------------------------------
SUBROUTINE ERRMSG(routine, status, pos)

  IMPLICIT NONE

  ! I/O
  CHARACTER(LEN=*), INTENT(IN)  :: routine
  INTEGER,          INTENT(IN)  :: status
  INTEGER,          INTENT(IN)  :: pos
! ------------------------------------------------------------------------------
```

It returns without any action, if the `status` equals zero. Otherwise it produces an error message printing the `status` flag and the position `pos`. Finally, the simulation is terminated.

variable name	number	meaning
MSGMODE_S	0	silent
MSGMODE_E	1	error messages only
MSGMODE_VL	2	little verbosity
MSGMODE_W	4	warning messages
MSGMODE_VM	8	medium verbosity
MSGMODE_I	16	all info messages

Table 3: List of verbosity stages

marker	number	meaning
RGMLE	0	error
RGMLEC	1	error continued
RGMLVL	2	little verbose
RGMLVLC	3	little verbose continued
RGMLW	4	warning
RGMLWC	5	warning continued
RGMLVM	6	medium verbose
RGMLVMC	7	medium verbose continued
RGMLI	8	information
RGMLIC	9	information continued

Table 4: List of message verbosity markers.

3.1.3.13 RGMSG

The submodel GRID uses a multi-level message system. `RGMSG` is an interface for four different message subroutines (see below). In this way the talkativeness of the mapping software can be easily changed. This is helpful, as during debugging phases a lot of information simplifies the troubleshooting. Otherwise, during long-term simulations a longish log-file output is awkward and costs unnecessary simulation time. The verbosity of the message system is set via the subroutine `MAIN_GRID_SET_MESSAGEMODE`. Six stages of verbosity exist. Table 3 lists the identifiers used from the user side to determine how many output is produced. The interface `RGMSG` is called with markers indicating how important the respective output is. Table 4 lists these identifiers.

In case of "error", "warning" or "info" the messages start with a respective identifier and afterwards print the string provided to the message subroutine. The "continued" specifiers indicate that the message is a continuation of an already started message, thus no identifier is printed and the text is indented.

Additionally, `RGMSG` terminates the simulation, if this is triggered by the optional argument lstop. If it is provided and `.TRUE.` the simulation will aborted after printing the message text. In case of error messages (identifier `RGMLE` or `RGMLEC`) the simulation will be aborted automatically. Only if `lstop = .FALSE.` this overrules the automatically produced abortion. In this way a continued error message is generated.

3.1.3.13.1 RGMSG_C

The main working horse is the subroutine `RGMSG_C`. It produces the above mentioned message types and forces the model to stop, if necessary.

```
! ----------------------------------------------------------------------------
SUBROUTINE RGMSG_C(routine, level, c, lstop)

#if defined(MPI)
  USE messy_main_grid_mpi,  ONLY: grid_abort
#endif

  IMPLICIT NONE

  ! I/O
  CHARACTER(LEN=*), INTENT(IN)            :: routine
  INTEGER,          INTENT(IN)            :: level
  CHARACTER(LEN=*), INTENT(IN)            :: c
  LOGICAL,          INTENT(IN), OPTIONAL  :: lstop
! ----------------------------------------------------------------------------
```

Input to the subroutine are the name of the calling subroutine (`routine`), the verbosity of the output (`level`, see Table 4), a string containing the error message itself (`c`), and the optional argument `lstop` indicating whether the simulation shall be terminated.

3.1.3.13.2 RGMSG_I

In addition to pure character output it might be required to print integer numbers.

Therefore the subroutine `RGMSG_I` allows to print an integer value (`i`) flanked by the character stings (`c1` and `c2`).

```
! ----------------------------------------------------------------------------
SUBROUTINE RGMSG_I(routine, level, c1, i, c2, lstop)

  IMPLICIT NONE

  ! I/O
  CHARACTER(LEN=*), INTENT(IN)            :: routine
  INTEGER,          INTENT(IN)            :: level
  CHARACTER(LEN=*), INTENT(IN)            :: c1
  INTEGER,          INTENT(IN)            :: i
  CHARACTER(LEN=*), INTENT(IN)            :: c2
  LOGICAL,          INTENT(IN), OPTIONAL  :: lstop
! ----------------------------------------------------------------------------
```

Internally, a character string is produced from the two strings and the integer value. This string is subsequently used as input to the routine `RGMSG_C`.

3.1.3.13.3 RGMSG_IA

If not only one integer, but an array of integers should be printed, the subroutine `RGMSG_IA` is called.

```
! ----------------------------------------------------------------------------
SUBROUTINE RGMSG_IA(routine, level, c1, i, c2, lstop)
```

```
  IMPLICIT NONE

  ! I/O
  CHARACTER(LEN=*),      INTENT(IN)            :: routine
  INTEGER,               INTENT(IN)            :: level
  CHARACTER(LEN=*),      INTENT(IN)            :: c1
  INTEGER, DIMENSION(:), INTENT(IN)            :: i
  CHARACTER(LEN=*),      INTENT(IN)            :: c2
  LOGICAL,               INTENT(IN), OPTIONAL  :: lstop
! ------------------------------------------------------------------------------
```

As in the subroutine `RGMSG_I` a character string is produced from the two character strings `c1` and `c2` and the integer vector. This in turn is used as input for the subroutine `RGMSG_C`.

3.1.3.13.4 RGMSG_R

Last but not least, the subroutine `RSMSG_R` enables the output of a single precision float value. As in `RSMSG_I` a single character string is created from the float value and the two input strings `c1` and `c2`, which serves as input for the subroutine call to `RGMSG_C`.

3.1.3.14 MAIN_GRID_SET_MESSAGEMODE

With this subroutine the verbosity of the error message system `RGMSG` is controlled.

```
! ------------------------------------------------------------------------------
SUBROUTINE MAIN_GRID_SET_MESSAGEMODE(SETVALUE)

  ! set MESSAGE mode

  IMPLICIT NONE

  INTEGER, INTENT(IN), OPTIONAL :: SETVALUE
! ------------------------------------------------------------------------------
```

If called without an argument, the highest possible verbosity is applied. Possible values are arbitrary sums of message mode markers (see Table 3).

3.1.3.15 NARRAYMAXVAL, NARRAYMINVAL

These functions provide the maximum or minimum value of the `dat` component for a given variable of type `t_narray`.

3.2 MESSY_MAIN_GRID

The module file `MESSY_MAIN_GRID` contains basically everything that is required for the definition and handling of the geohybrid grids. In the module header the structure for geohybgrid grids of type `t_geohybgrid` is defined. As in a model simulation a large variety of geohybrid grids may be required, a concatenated list of variables of type `t_geohybgrid`, GEOHYBGRIDLIST, of type `t_geohybgrid_list` is constructed.

3.2.1 TYPE t_geohybgrid

Due to different grid structures a geohybrid grid can be defined in many different ways. The structure `t_geohybgrid` contains all information required for the submodel GRID.

```
! ----------------------------------------------------------------------------
TYPE t_geohybgrid
    CHARACTER(LEN=GRD_MAXSTRLEN) :: name         ! grid name
    INTEGER                      :: ID     = -99 ! GRID ID
    CHARACTER(LEN=GRD_MAXSTRLEN) :: file         ! path/filename
    INTEGER                      :: t            ! time step
    TYPE(t_ncatt)                :: att          ! attribute
    REAL(dp), DIMENSION(4,2)     :: ranges = RGEMPTY
    !
    ! MINIUM / MAXIMUM GEOGRAPHICAL EXTENSION
    REAL(dp), DIMENSION(2,2)     :: minmaxlonlat = RGEMPTY
    INTEGER,  DIMENSION(2)       :: start        = -99
    INTEGER,  DIMENSION(2)       :: count        = -99
    LOGICAL                      :: lonc   = .TRUE.
    TYPE (t_ncvar)               :: lonm, latm, hyam, hybm, timem !mid-layer
    TYPE (t_ncvar)               :: loni, lati, hyai, hybi        !interface
    TYPE (t_ncvar)               :: ps, p0
    ! curved grid variables
    ! geographical coordinates
    LOGICAL                      :: clonc  = .TRUE.
    TYPE (t_ncvar)               :: clonm, clatm, cloni, clati
    ! rotated coordinates
    LOGICAL                      :: rlonc  = .TRUE.
    TYPE (t_ncvar)               :: rlonm, rlatm, rloni, rlati
    ! rotated pole for rotated coordinates
    TYPE (t_ncvar)               :: pollon, pollat, polgam
 END TYPE t_geohybgrid
! ----------------------------------------------------------------------------
```

A variable of type `t_geohybgrid` gets a unique name (`name`) and a unique ID (`ID`). If the definition of a geohybrid grid is read from a file, the file name and path (`file`) and the time step (`t`) (i.e., the hyperslice along the unlimited id) are required. For the more in depth description of the grid, an attribute (`att`) can be provided. Depending on the basemodel, some variables have to be limited in certain `ranges`. The structure component `ranges` contains for all four space dimensions lower and upper bounds. The structure component `minmaxlonlat` is required for the subroutine `REDUCE_INPUT_GRID`

(see Sect. 3.3.12) to check whether both grids are defined over the same domain. The structure components `start` and `count` are also required for the input grid reduction.

Horizontal coordinates of the grid can be defined on grid mid-points (variable name ends with `m`) and/or on interfaces (variable name ends with `i`). Depending on the actual grid, different definitions of the horizontal coordinates apply. The reference system is always based on geographical coordinates. Currently, three different sets for the definition of the horizontal coordinates are available in `t_geohybgrid`:

1. *rectangular geographical coordinates:* `lonm`, `latm`, `loni` and `lati`
   These variables of type `t_ncvar` are defined for geographically-rectangular grids. Thus these variables contain geographical coordinate information and correspond to only one dimension, i.e., a simple longitude or latitude axis. Additionally, the logical variable `lonc` indicates whether the longitude axis is a modulo axis.

2. *curvilinear coordinates:* `clonm`, `clatm`, `cloni`, `clati`
   These are coordinates which are not rectangular in geographical coordinates, e.g. a rotated regional domain or an ocean model grid. The variables contain geographical coordinates, but, as they are not rectangular in geographical coordinates, they comprise coordinate information for each single grid point. Additionally, the logical variable `clonc` indicates whether the longitude axis is a modulo axis.

3. *rotated rectangular coordinates:* `rlonm`, `rlatm`, `rloni`, `rlati`:
   These coordinates are again rectangular, but they do not contain geographical coordinates. These structure components are useful for rotated grids, as these can be handled similar to geographical coordinates as long as only grids with the same rotation are transformed into each other. The rotation of the grid in reference to the geographical system is defined by the longitude, the latitude and the rotation angle for the north pole, i.e. `pollon`, `pollat` and `polgam`, respectively. Additionally, the logical variable `rlonc` indicates whether the longitude axis is a modulo axis.

The vertical axis is defined via hybrid coefficients (on box mids and interfaces, `hyam`,`hybm`,`hyai`,`hybi`, respectively) and the surface pressure `ps` and a reference pressure `p0`.

`timem` defines the point in time the grid is associated to.

3.2.2 FUNCTIONS and SUBROUTINES for grid handling

The module `MESSY_MAIN_GRID` contains routines to handle variables of the above described type:

3.2.2.1 INIT_GEOHYBGRID

This subroutines initialises all components of the variable `grid` of type `t_geohybgrid`, i.e., all components are set to default values.

3.2.2.2 COPY_GEOHYBGRID

This subroutine copies one variable of type `t_geohybgrid` to another variable of the same type.

At the beginning of the subroutine, the destination variable is re-initialised. Therefore, all information stored in the variable upon calling this subroutine are lost.

3.2.2.3 PRINT_GEOHYBGRID

This subroutine prints components of the variable `grid` of type `t_geohybgrid` into the log-file. It is not called during model simulation, as it produces an enormous amount of output. Nevertheless, it is a useful tool during code implementation, to test if the grid is defined as expected.

3.2.2.4 IMPORT_GEOHYBGRID

This subroutine imports all components of a variable of type `t_geohybgrid` from a netCDF file, mostly by calling `IMPORT_NCVAR` (see Sect. 3.1.2.6.3).

```
!------------------------------------------------------------------------------
SUBROUTINE IMPORT_GEOHYBGRID(grid, pstart, pcount)

  ! read geohybgrid from netcdf file
  ! NO CHECKING IN THIS ROUTINE !!!

  IMPLICIT NONE

  ! I/O
  TYPE (t_geohybgrid),   INTENT(INOUT)        :: grid
  INTEGER, DIMENSION(2), INTENT(IN), OPTIONAL :: pstart
  INTEGER, DIMENSION(2), INTENT(IN), OPTIONAL :: pcount
!------------------------------------------------------------------------------
```

Here, `pstart` and `pcount` define the hyperslice of data to be read. `pstart` indicates the position at which the reading should start, `pcount` how many data points should be read from `pstart` on. This is used to reduce the size of the input grid and thus the required memory. This is of special interest, if GRID is used in parallel domain decomposition. `pstart` and `pcount` contain 2 entries each in order to limit the data in longitude and latitude direction. If the data is only 1D in the horizontal, the second entry has no meaning.

3.2.2.5 EXPORT_GEOHYBGRID

This subroutine exports the components of a variable `grid` of type `t_geohybgrid` to a netCDF file. The filename is given by the component `grid%file`. The components of type `t_ncvar` are written to the file by calling `EXPORT_NVCAR`.

3.2.2.6 NEW_GEOHYBGRID

During more complex model simulations, the definition of serveral geohybrid grids may be necessary. To make them universally accessible and searchable, new grids are added to a concatenated list called `GEOHYBGRIDLIST`. The subroutine `NEW_GEOHYBGRID` adds a new entry to this list. A new list element can be provided either by a variable of type `t_geohybgrid`, which has been defined beforehand somewhere else in the code (`NEW_GEOHYBGRID_BY_GRID`) or by providing the components of the grid (`NEW_GEOHYBGRID_BY_COMPONENTS`). Internally, `NEW_GEOHYBGRID_BY_GRID` calls `NEW_GEOHYBGRID_BY_COMPONENTS` passing the individual grid components.

```
! ============================================================================
SUBROUTINE NEW_GEOHYBGRID_BY_COMPONENTS( status, id, name              &
                                       , lonm, latm, hyam, hybm, timem &
                                       , loni, lati, hyai, hybi,       &
                                       , clonm, clatm, cloni, clati    &
                                       , rlonm, rlatm, rloni, rlati    &
                                       , ps,    p0,    file,  t        &
                                       , ranges, minmaxlonlat          &
                                       , pollon, pollat, polgam, col)

  ! define new geohybgrid
  ! INPUT:
  !  -  components of geohybgrid
  ! OUTPUT:
  !  - id:    the id of the newly defined grid

  IMPLICIT NONE

  ! I/O
  INTEGER,                 INTENT(OUT)          :: status
  INTEGER,                 INTENT(OUT)          :: id
  CHARACTER(LEN=*),        INTENT(IN), OPTIONAL :: name
  ! mid-layer
  TYPE(t_ncvar),           INTENT(IN), OPTIONAL :: lonm
  TYPE(t_ncvar),           INTENT(IN), OPTIONAL :: latm
  TYPE(t_ncvar),           INTENT(IN), OPTIONAL :: hyam
  TYPE(t_ncvar),           INTENT(IN), OPTIONAL :: hybm
  TYPE(t_ncvar),           INTENT(IN), OPTIONAL :: timem
  ! interface layer
  TYPE(t_ncvar),           INTENT(IN), OPTIONAL :: loni
  TYPE(t_ncvar),           INTENT(IN), OPTIONAL :: lati
  TYPE(t_ncvar),           INTENT(IN), OPTIONAL :: hyai
  TYPE(t_ncvar),           INTENT(IN), OPTIONAL :: hybi
  ! curvilinear
  TYPE(t_ncvar),           INTENT(IN), OPTIONAL :: clonm
  TYPE(t_ncvar),           INTENT(IN), OPTIONAL :: clatm
  TYPE(t_ncvar),           INTENT(IN), OPTIONAL :: cloni
  TYPE(t_ncvar),           INTENT(IN), OPTIONAL :: clati
  ! rotated curvilinear
  TYPE(t_ncvar),           INTENT(IN), OPTIONAL :: rlonm
  TYPE(t_ncvar),           INTENT(IN), OPTIONAL :: rlatm
  TYPE(t_ncvar),           INTENT(IN), OPTIONAL :: rloni
  TYPE(t_ncvar),           INTENT(IN), OPTIONAL :: rlati
  ! pressure
  TYPE(t_ncvar),           INTENT(IN), OPTIONAL :: ps
  TYPE(t_ncvar),           INTENT(IN), OPTIONAL :: p0
  ! rotated north pole
  TYPE(t_ncvar),           INTENT(IN), OPTIONAL :: pollon
```

```
  TYPE(t_ncvar),            INTENT(IN), OPTIONAL :: pollat
  TYPE(t_ncvar),            INTENT(IN), OPTIONAL :: polgam
  ! unstructured grid
  TYPE(t_ncvar),            INTENT(IN), OPTIONAL :: col
  ! if required file name for imported grid
  CHARACTER(LEN=*),         INTENT(IN), OPTIONAL :: file
  ! if required time step in file of imported grid
  INTEGER,                  INTENT(IN), OPTIONAL :: t
  REAL(dp), DIMENSION(4,2), INTENT(IN), OPTIONAL :: ranges
  REAL(dp), DIMENSION(2,2), INTENT(IN), OPTIONAL :: minmaxlonlat

  ! LOCAL
  TYPE(t_geohybgrid_list), POINTER :: gi => NULL()
  TYPE(t_geohybgrid_list), POINTER :: ge => NULL()
  TYPE(t_geohybgrid),      POINTER :: lgrid
  INTEGER                          :: cid ! grid id returned by compare
! ================================================================================
```

The subroutine has only two mandatory arguments: the output variables `status` indicating success, error or that the grid exists already, and the `ID` of the newly defined (or the already defined, identical) grid. All other structure components of `t_geohybgrid` are optional arguments, thus only those meaningful for the desired grid have to be provided. While the `ID` is used to identify geohybrid grids unambiguously internally in GRID, the name of the grid is required to identify the grid for external use (e.g., in IMPORT_GRID, if in the namelist a special target grid is requested). Thus, if the `name` is present and non-empty the definition of a grid of this name is forced, even of all other components of the grid are equal to another already defined grid. If the `name` is not present or empty the grid will be named genericly.

At the beginning, the subroutine cycles through the list of already defined grids calling `COMPARE_TO_GRID` (see Sect. 3.2.2.7) for each entry.

- The cycling is continued as long as `COMPARE_TO_GRID` returns with an negative ID, indicating that the grids are not equal.
- If the grids are equal and `name` is not present or empty, `ID` is set to the ID of the already defined grid and `status` is set to `"1"` indicating, that no new grid has been defined.
- If the grids are equal and a name is present, non-empty and not equivalent to the grid, the cycling of the grid list is continued.

If a non-empty `name` is argument to the subroutine, an additional check is performed during the cycling of the grid list: in order to make the grids later on uniquely identifyable also by name, the names of the grid must be different. Therefore, it is checked whether the names are equal, if the grids differ. Otherwise the simulation is terminated with an error.

If the new grid is not equal to one of the grids in the list, a new list entry is created and the subroutine returns the new `ID` of the list entry and the `status` is set to zero, indicating success, i.e., the definition of a new list entry. In case that a non-empty `name` is argument to the subroutine, this name is used, otherwise the subroutine generates a grid name consisting of the string 'GENERICGRID' and a five digits long number.

3.2.2.7 COMPARE_TO_GRID

This subroutine compares a list of grid components to a provided variable of type `t_geohybgrid`.

```
! ============================================================================
SUBROUTINE COMPARE_TO_GRID(id, grid, lonm,  latm,  hyam,  hybm, timem  &
                                   , loni,  lati,  hyai,  hybi,        &
                                   , clonm, clatm, cloni, clati        &
                                   , rlonm, rlatm, rloni, rlati        &
                                   , ps,    p0,    file,  t,    ranges &
                                   , minmaxlonlat, pollon, pollat, polgam)
! ============================================================================
```

Mandatory arguments are only the output variable `ID`, which, in case of identical grids, contains the grid ID, and the pointer to the grid, which should be compared to the components.

In the remainder of the subroutine, each present component is compared to the respective component of `grid`. If they differ, the subroutine returns with ID set to `-99`, indicating that the grid components are not equal to the provided components.

If the component is not present, but defined for `grid` the subroutine also returns with ID set to `-99` indicating that the grid components are not fully equal to the provided components.

At the end, after the comparison of all components of the variable type `t_geohybgrid`, all components of the grids are equal and the output variable `ID` is set to the ID of `grid`.

Note: there are two components that are not compared. First the ID, as this is simply the identifier of the grid in the list and second, the name is not compared, as it is often generated generic by `NEW_GEOHYBGRID`. If the name should also be tested this needs to be done "by hand" (as in `NEW_GEOHYBGRID`).

3.2.2.8 LOCATE_GEOHYBGRID

During more complex model simulations it might be of interest to locate and access a certain grid in the list created by `NEW_GEOHYBGRID`. A grid is uniquely identifyable by its ID and by its name. Therefore the subroutine interface `LOCATE_GEOHYBGRID` comprises two subroutines: `LOCATE_GEOHYBGRID_BY_ID` and `LOCATE_GEOHYBGRID_BY_NAME`.

3.2.2.8.1 LOCATE_GEOHYBGRID_BY_ID

This subroutine locates a variable of type `t_geohybgrid` based on an ID within the list of geohybrid grids (`GEOHYBGRIDLIST`).

```
!------------------------------------------------------------------------------
SUBROUTINE LOCATE_GEOHYBGRID_BY_ID(status, ID, pgrid, grid)

  ! search grid according to the given ID
  ! OUTPUT:
  !   - grid:  the grid itself
  !   - pgrid: a pointer to the grid

  IMPLICIT NONE
```

```
  ! I/O
  INTEGER, INTENT(OUT)                    :: status
  INTEGER, INTENT(IN)                     :: ID
  TYPE(t_geohybgrid),   POINTER, OPTIONAL :: pgrid
  TYPE(t_geohybgrid),            OPTIONAL :: grid
!------------------------------------------------------------------------------
```

Input to the subroutine `LOCATE_GEOHYBGRID_BY_ID` is only an ID. The list of defined grids is cycled until the ID is found. Depending on the arguments, a pointer `pgrid` is associated to the corresponding list element or the content of the list element is copied to the output variable `grid`. In all cases a `status` flag provides information about success or failure of the subroutine.

3.2.2.8.2 LOCATE_GEOHYBGRID_BY_NAME

This subroutine locates a variable of type `t_geohybgrid` based on the grid name within the list of geohybrid grids (`GEOHYBGRIDLIST`).

```
!------------------------------------------------------------------------------
SUBROUTINE LOCATE_GEOHYBGRID_BY_NAME(status, name, pgrid, grid, ID)

  ! search grid according to the given name
  ! OUTPUT:
  !   - grid:  the grid itself
  !   - pgrid: a pointer to the grid
  !   - ID:    the grid ID

  USE messy_main_constants_mem, ONLY: STRLEN_MEDIUM

  IMPLICIT NONE

  INTRINSIC :: ADJUSTL, ASSOCIATED, LEN_TRIM, TRIM

  ! I/O
  INTEGER,            INTENT(OUT)             :: status
  CHARACTER(LEN=*),   INTENT(IN)              :: name
  TYPE(t_geohybgrid), POINTER,       OPTIONAL :: pgrid
  TYPE(t_geohybgrid), INTENT(INOUT), OPTIONAL :: grid
  INTEGER,            INTENT(OUT),   OPTIONAL :: ID
!------------------------------------------------------------------------------
```

Input to the subroutine `LOCATE_GEOHYBGRID_BY_NAME` is only the name of the grid. The list of defined grids is cycled until the grid with the name is found. Depending on the arguments, a pointer `pgrid` is associated to the corresponding list element, or the content of the list element is copied to the output variable `grid`. Furthermore, on demand this subroutine hands back the ID of the grid. In all cases a `status` flag provides information about success or failure of the subroutine.

3.2.2.9 CLEAN_GEOHYBGRID_LIST

At the end of the simulation the concatenated list of geohybgrid grids needs to be deleted. This is done by the subroutine `CLEAN_GEOHYBGRID_LIST`.

3.2.2.10 Function GRID_ERROR

Within the subroutines of the submodel GRID consistency checks are performed and status flags are set accordingly. These integer numbers are converted to meaningful error messages by the function `GRID_ERROR`.

```
!------------------------------------------------------------------------------
CHARACTER(LEN=256) FUNCTION GRID_ERROR(status)

  ! This subroutine provides an error string for a given status

  IMPLICIT NONE

  INTEGER, INTENT(IN) :: status

!------------------------------------------------------------------------------
```

The function `GRID_ERROR` produces the error string for a given `status` flag.

3.3 MESSY_MAIN_GRID_TRAFO

The module MESSY_MAIN_GRID_TRAFO contains routines necessary to operate on grids. These routines are mostly applied during the transformation / mapping process in REGRID_CONTROL and SCRIP_CONTROL.

3.3.1 SWITCH_GEOHYBGRID

This subroutine deletes dimensions and the respective components from a variable of type t_geohybgrid.

```
!------------------------------------------------------------------------------
SUBROUTINE SWITCH_GEOHYBGRID(g, lx, ly, lz)

  ! this subroutine initializes (=deletes) dimensions of a geohybgrid,
  ! if respective logicals are true:
  ! - lx = .FALSE: : re-initialise lonm and loni
  ! - ly = .FALSE: : re-initialise latm and lati
  ! - lz = .FALSE: : re-initialise vertical definition

  IMPLICIT NONE

  ! I/O
  TYPE(t_geohybgrid), INTENT(INOUT) :: g
  LOGICAL           , INTENT(IN)    :: lx, ly, lz
!------------------------------------------------------------------------------
```

The logicals lx, ly, lz indicate which dimensions are erased. If lz=.FALSE., the components hyai, hybi, hyam, hybm, p0 and ps are re-initialised. For lx=.FALSE., lonm and loni, and for ly=.FALSE., latm and lati are re-initialised. The rotated and curvilinear components (clonm, cloni, clatm, clati, rlonm, rloni, rlatm, rlati) are deleted, if either lx or ly, or both are .FALSE..

3.3.2 CHECK_GEOHYBGRID

The subroutine CHECK_GEOHYBGRID checks the components of a geohybrid grid for consistency. The subroutine terminates the simulation if

- the number of dimensions is larger than 1
  - for the horizontal components lonm, loni, latm, lati,
  - for the time variable timem,
  - for the hybrid coefficients hyai, hybi, hyam, hybm.
- for defined hyai and hybi the lengths of the dimension are not equal.
- for defined hyam and hybm the lengths of the dimension are not equal.
- for defined hyam and hyai the length of the dimension of hyai is not exactly larger by one as the dimension length of hyam.

- for defined `hybm` and `hybi` the length of the dimension of `hybi` is not exactly larger by one as the dimension length of `hybm`.
- the number of dimensions of the surface pressure component `ps` is larger than 3 (longitude, latitude and time).
- no meaningful `ranges` are set in the special cases where the dimension length of `lonm`, `latm`, `hyam` or `hybm` equals 1.

3.3.3 SORT_GEOHYBGRID

This subroutine sorts the inidividual grid components from smallest to largest. This subroutine can only be applied of the dimensions are independent of each other. Therefore it is not applicable for curvilinear grids. This subroutine is freuqently used in NREGRID, as for NREGRID the independence of the dimensions is a prerequisite. It is not called for SCRIP application, as SCRIP deals with more general grids.

```
!------------------------------------------------------------------------------
SUBROUTINE SORT_GEOHYBGRID(gi, go, gx, reverse)

  IMPLICIT NONE

  ! I/O
  TYPE (t_geohybgrid), INTENT(IN)    :: gi  ! INPUT GRID
  TYPE (t_geohybgrid), INTENT(OUT)   :: go  ! OUTPUT GRID
  TYPE (t_geohybgrid), INTENT(INOUT) :: gx  ! INDEX 'GRID'
  LOGICAL, OPTIONAL,   INTENT(IN)    :: reverse
!------------------------------------------------------------------------------
```

Within the subroutine the grid components `lonm`, `loni`, `latm`, `lati`, `timem`, `p0`, `hyai`, `hyam`, `hybi`, `hybm` and `ps` are sorted mainly by calling the subroutine `SORT_NARRAY` from module `MESSY_MAIN_GRID`.

3.3.4 H2PSIG

This subroutine calculates sigma or pressure levels from hybrid pressure coordinates.

```
!------------------------------------------------------------------------------
SUBROUTINE H2PSIG(psig, hya, hyb, ps, p0, lp)

  IMPLICIT NONE

  ! I/O
  TYPE (t_narray), INTENT(INOUT) :: psig
  TYPE (t_narray), INTENT(IN)    :: hya, hyb, ps, p0
  LOGICAL        , INTENT(IN)    :: lp   ! .true.  -> pressure axis
                                         ! .false. -> dimensionless axis
!------------------------------------------------------------------------------
```

Input of the subroutine are variables of type `t_narray` for the hybrid coefficients (`hya`, `hyb`), the surface (`ps`) and the reference (`p0`) pressure. Additionally the flag `lp` indicates, whether the pressure `psig` should be given in sigma or in pressure coordinates.

Dependent on the definition status of the variables, the vertical coordinate as pressure or sigma coordinate is calculated in different ways:

- Hybrid pressure levels (`hya` and `hyb` are both defined):
  Pressure coordinates (`lp=.TRUE.`) are calculated as

  $$psig = hya * p0 + hyb * ps, \tag{1}$$

  sigma coordinates (`lp=.FALSE.`) as

  $$psig = (hya * p0 + hyb * ps)/ps. \tag{2}$$

- Sigma levels (`hya` is undefined, `hyb` defined):
  Pressure coordinates (`lp=.TRUE.`) are calculated as

  $$psig = hyb * ps \tag{3}$$

  and sigma coordinates (`lp=.FALSE.`) as

  $$psig = hyb/hyb_{max} \tag{4}$$

  with hyb_{max} the largest entry of `hyb`.

- Constant pressure levels (`hya` is defined, `hyb` is undefined):
  Pressure coordinates (`lp=.TRUE.`) are calculated as

  $$psig = hya * p0 \tag{5}$$

  and sigma coordinates (`lp=.FALSE.`) as

  $$psig = hya/hya_{max} \tag{6}$$

  with hya_{max} the largest entry of `hya`.

3.3.5 COMPLETE_GEOHYBGRID

This subroutine checks a variable of type `t_geohybgrid`, if the variables on mid-points and interfaces are consistently defined. Additionally, if only mid-point or interface variables are defined, the missing one is calculated from the other.

```
! ------------------------------------------------------------------------------
SUBROUTINE COMPLETE_GEOHYBGRID(g, gx, oranges)

  IMPLICIT NONE

  ! I/O
  TYPE (t_geohybgrid),      INTENT(INOUT)           :: g
  TYPE (t_geohybgrid),      INTENT(INOUT), OPTIONAL :: gx   ! SORT INDICES
  REAL(DP), DIMENSION(4,2), INTENT(IN)   , OPTIONAL :: oranges
! ------------------------------------------------------------------------------
```

Here, g is the input grid of type t_geohybgrid and gx is the index grid containing the information how to sort (back) the components of g (see Sect. 3.3.3).

For each of the variable pairs (lonm, loni), (latm, lati), (hyam, hyai), (hybm, hybi) the same sequence of four checking subroutines IMMI_NARRAY, IMMI_NCVAR, IMMI_NARRAY_IDX, IMMI_NCVAR, is called. For the component pairs (clonm, cloni) and (clatm, clati) only IMMI_NARRAY with ltestonly =.TRUE. is called as the other routines are not applicable to curvilinear coordinates. For the rotated coordinates (rlonm, rloni) and (rlatm, rlati) the two subroutines IMMI_NARRAY and IMMI_NCVAR plus the additional subroutines IMMI_CLONI_CLATI and IMMI_CLONM_CLATM, respectively, are called.

3.3.5.1 IMMI_NARRAY

This subroutine checks the consistency of a variable pair of type t_narray and dependent on the verbosity of the output, writes each diagnostic step into the log-file.

- Everything is ok, if both variables are undefined.
- If both are defined, they have to be of the same type. Otherwise, an information about the number of dimensions is written to the log-file and the simulation is stopped by calling RGMSG with an error flag. Finally, the length of the dimension of the interface variable has to be exactly greater by one as the length of the dimension of the mid-point variable.
- If one of the variables is undefined, the other defined, the undefined one is calculated from the other. This is only applicable for rectangular grids. Thus, this calculation can be skipped by setting the input logical switch ltestonly .TRUE..

3.3.5.2 IMMI_NARRAY_IDX

If the input grid of type t_geohybgrid upon the call of COMPLETE_GEOHYBGRID is already sorted (see Sect. 3.3.3) an additional grid variable gx has to be argument of the call containing the information how to sort back the grid components to the order of the original grid. This information is also required for the the components added by IMMI_NARRAY to the original grid g. Thus, the components added to g also need to be added to gx. This is done by the subroutine IMMI_NARRAY_IDX.

3.3.5.3 IMMI_NCVAR

The subroutines IMMI_NARRAY and IMMI_NARRAY_IDX only analyse the data structure components of the variables, i.e. they determine the dimensions and calculate missing components if possible. The additional information required for a full variable of type t_ncvar, i.e., the name, id, and attributes etc., are added by the subroutine IMMI_NCVAR.

3.3.5.4 IMMI_CLONM_CLATM

This subroutine calculates clonm and clatm from the rotated coordinates, if rlonm, rlatm are defined. For this calculation the definition of the rotated pole (pollon, pollat and polgam) and the two conversion subroutines RLAROT2RLA and PHIROT2PHI (Sect. 3.3.13) are used.

3.3.5.5 IMMI_CLONI_CLATI

This subroutine calculates cloni, clati from the rotated coordinates, if rloni and rlati are defined. For this calculation the definition of the rotated pole (pollon, pollat and polgam) and the two conversion subroutines RLAROT2RLA and PHIROT2PHI (Sect. 3.3.13) are used.

3.3.5.6 RNGADJ_NARRAY

This subroutine adjusts the ranges of the newly created coordinate components according to `oranges` or, if `oranges` is not present, to `g%ranges`.

3.3.6 CHECK_NCVAR_ON_GEOHYBGRID

This subroutine checks if a variable `var` of type `t_ncvar` is defined on a grid `g` of type `t_geohybgrid`.

```
!------------------------------------------------------------------------------
SUBROUTINE CHECK_NCVAR_ON_GEOHYBGRID(var, g, dims, axes, ok)

  IMPLICIT NONE

  ! I/O
  TYPE (t_ncvar)     , INTENT(IN)  :: var    ! variable
  TYPE (t_geohybgrid), INTENT(IN)  :: g      ! grid
  INTEGER, DIMENSION(:), POINTER   :: dims   ! order of g-dims in var
  INTEGER            , INTENT(OUT) :: axes(3)! dimension no. of lon->lat->lev
  LOGICAL            , INTENT(OUT) :: ok     ! conform ?
!------------------------------------------------------------------------------
```

Output of the subroutine are

- a 1D pointer `dims` dimensioned by the number of dimensions of `var`. It associates the i-th dimension of variable `var` with the j-th dimension of `g`.
- an integer array `axes` of length 3 identifying the i-th dimension of `var` as longitude, latitude and level axis (in this order).
- the logical `ok` which is `.TRUE.`, if the variable definition is conform with the grid definition.

Example: if in the grid the dimension variables are allocated in the order (lon, lat, lev, num) and in `var` in the order (lon, num, lev, lat) this results in `dims =(1,0,3,2)` and `axes = (1,4,3)`.

3.3.7 SORT_GEOHYBGRID_NCVAR

This subroutine sorts a variable `var` according to the index information provided by a geohybrid grid with index information `gx`. The newly sorted variable `svar` is output of the subroutine.

```
!------------------------------------------------------------------------------
SUBROUTINE SORT_GEOHYBGRID_NCVAR(var, gx, axes, svar, reverse)

  IMPLICIT NONE

  ! I/O
  TYPE (t_ncvar)     , INTENT(IN)           :: var     ! input variable
  TYPE (t_geohybgrid), INTENT(IN)           :: gx      ! hybrid grid with  ...
                                                       ! ... index information
```

```
  INTEGER            , INTENT(IN)           :: axes(3) ! lon,lat,lev dim. no.
  TYPE (t_ncvar)     , INTENT(INOUT)        :: svar    ! sorted variable
  LOGICAL            , INTENT(IN), OPTIONAL :: reverse
!------------------------------------------------------------------------------
```

If `reverse` is given and `.TRUE.` the variable is reordered to its original order. Additional, mandatory input of the subroutine is the information of the order of the dimensions in the variable (`axes`). This information is obtained by calling `CHECK_NCVAR_ON_GEOHYBGRID` prior to the call to this subroutine.

3.3.8 PACK_GEOHYBGRID_NCVAR

This subroutine (un-)packs a variable `vi` of type `t_ncvar` according to the axes and dimension information provided by the subroutine `CHECK_NCVAR_ON_GEOHYBGRID`.

```
!------------------------------------------------------------------------------
SUBROUTINE PACK_GEOHYBGRID_NCVAR(vi, dims, axes ,vo, reverse)

  IMPLICIT NONE

  ! I/O
  TYPE (t_ncvar)                , INTENT(IN)    :: vi   ! input variable
  INTEGER, DIMENSION(:)         , INTENT(IN)    :: dims
  INTEGER                       , INTENT(IN)    :: axes(3)
  TYPE (t_ncvar)                , INTENT(INOUT) :: vo   ! output variable
  LOGICAL, OPTIONAL             , INTENT(IN)    :: reverse
!------------------------------------------------------------------------------
```

The variable name of the packed variable is labeled by adding `_pd` at the end of the original variable name. The packing algorithm permutes the data in the 1d storage format according to transposed dimensions. These transposed dimensions are ordered such, that the invariant and unused dimensions are stored at the end.

3.3.9 BALANCE_GEOHYBGRID

Parameter of this subroutine are one input and one output grid, `gi` and `go`, respectively.

```
!------------------------------------------------------------------------------
SUBROUTINE BALANCE_GEOHYBGRID(gi, go)

  IMPLICIT NONE

  ! I/O
  TYPE (t_geohybgrid), INTENT(INOUT) :: gi      ! input grid
  TYPE (t_geohybgrid), INTENT(INOUT) :: go      ! output grid
!------------------------------------------------------------------------------
```

All coordinate variables[2], that do not exist in `go` but in `gi` are copied from `gi` to `go`.

[2] lonm, latm, loni, lati, clonm, clatm, cloni,clati, rlonm, rlatm, rloni, rlati, hyam, hybm, hyai, hybi

3.3.10 BALANCE_GEOHYBGRID_NCVAR

This subroutine dimensions a variable `varo` of type `t_ncvar` in such a way, that the three spacial axes are dimensioned according to a given output grid `go`, while the invariant dimensions are allocated as in the input variable `vari`.

```
!-------------------------------------------------------------------------------
SUBROUTINE BALANCE_GEOHYBGRID_NCVAR(vari, axes, go, varo, lrgz)

  USE messy_main_grid_netcdf,  ONLY: NULL_XTYPE

  IMPLICIT NONE

  ! Note: go must already be 'balanced'

  ! I/O
  TYPE(t_ncvar)     , INTENT(IN)    :: vari    ! input variable
  INTEGER           , INTENT(IN)    :: axes(3) ! dim.no of lon, lat, lev
  TYPE(t_geohybgrid), INTENT(IN)    :: go      ! output grid
  TYPE(t_ncvar)     , INTENT(INOUT) :: varo    ! output variable
  LOGICAL           , INTENT(IN), OPTIONAL :: lrgz
!-------------------------------------------------------------------------------
```

The subroutine parameter `axes` is required to associate the spacial dimensions to each other. Additionally, the optional parameter `lrgz` allows for keeping the vertical dimension. In the case `lrgz=.FALSE.` the vertical axis is treated as invariant axis.

3.3.11 BALANCE_GEOHYBGRID_TIME

The subroutine `BALANCE_GEOHYBGRID_TIME` adjusts the time axes of the two grids `gi` and `go`.

```
!-------------------------------------------------------------------------------
SUBROUTINE BALANCE_GEOHYBGRID_TIME(gi, go, lint)

  IMPLICIT NONE

  ! I/O
  TYPE (t_geohybgrid),    INTENT(INOUT) :: gi, go
  LOGICAL,                INTENT(IN)    :: lint
!-------------------------------------------------------------------------------
```

If `lint = .TRUE.` the time information of the input grid is used for the output grid. If `lint = .FALSE.` the output grid time information is used for the input grid. The balancing is performed for the time axis in the pressure component of the grid and for the time axis of the grids themselves.

3.3.12 REDUCE_INPUT_GRID

For interpolation of off-line data, the domain of the data to be remapped can be much larger as the target domain. An example is a global emission field, which should be remapped to a regional domain. Here the data read in from the file could be reduced to those covering the regional domain. This becomes even more efficient, if the regional model domain is decomposed by domain decomposition among serveral parallel tasks.

Therefore the subroutine `REDUCE_INPUT_GRID` reduces an input grid to the required size determined by the target domain. In a second read-cycle the fields are then read on the reduced grid.

Parameter to the subroutine are the input (or source) grid `gs`, which will be redefined in this subroutine, and the target (or destination) grid `gd`, which is required for a meaningful reduction of the input grid.

The subroutine works as follows:

1. Check, if a longitudinal dimension is defined.
2. Copy source and destination longitude to a local array and convert both to type double.
3. Check, if the two longitude axes are defined in the same way, i.e. in the interval `[-180,180]` or `[0,360]`; if not, both longitudinal axes are converted to `[0,360]`.
4. Check, if the source grid is larger than the target grid.
5. Repeat the above steps (1,2,4) for the latitude axes.
6. Determine the maximum/minimum longitude/latitudes (`lonmin`, `lonmax`, `latmin` and `latmax` that should be covered by the source grid by widening the destination grid by 2 times the grid spacing (`2*dlon` or `2*dlat`) at all sides. This domain is called the extended domain in the following.
7. Loop along all longitudes and latitudes to find the lowest and highest indices, at which the source grid boxes are located within the extended domain. The start indices are stored in the structure component `gs%start(1)` for the longitudes and `gs%start(2)` for the latitudes, respectively. For a curvilinear grid the two variables are equal. In addition to the start indices the counts are saved in the components `gs%count(1)` and `gs%count(2)` for longitudes and latitudes, respectively. They are determined as difference of the end and the start indices.
8. If the longitude axis of the grid is reduced after this procedure, the longitudinal axis is longer a modulo axis. Thus `gs%lonc`, `gs%clonc` and `gs%rlonc` are set `.FALSE.`.

Finally, some local variables are re-initialised to release the memory.

3.3.13 Conversion of rotated to geographical cooordinates and vise versa

For many applications, especially the mapping of different grids, different types of horizontal coordinates are required. The four functions `PHIROT2PHI`, `PHI2PHIROT`, `RLAROT2RLA` and `RLA2RLAROT` provide the functionality to convert longitudes (or lambda) and latitudes (or phi) from one rotated system to the other. The functions have been adopted from the COSMO model code. All four functions require four to five input parameters:

- the source latitude

- the source longitude
- the latitude of the north pole of the rotated system (`polphi`)
- the longitude of the north pole of the rotated system (`pollam`)
- optionally, the angle between the north poles of the rotated systems (`polgam`). If `polgam` is not present, the other system is the geographical system.

3.3.13.1 PHI2PHIROT

This function calculates the target (if `polgam` is not present, geographic) latitude.

3.3.13.2 PHIROT2PHI

This function calculates the target (i.e., rotated) latitude.

3.3.13.3 RLAROT2RLA

This function calculates the target (if `polgam` is not present, geographic) longitude.

3.3.13.4 RLA2RLAROT

This function calculates the target (i.e., rotated) longitude.

3.3.14 Wind vector conversion between grids: UVROT2UV_VEC, UV2UVROT_VEC, UVROT2UV_VEC_JLOOP

In contrast to the coordinate transformation of scalars, vector variables need to be converted in a different way. Therefore the subroutines `UVROT2UV_VEC` and `UV2UVROT_VEC` convert the wind vector defined by the variables `u` and `v` from the rotated system to the geographical system and vice versa.

Parameter of the subroutines are the 2D (`u`, `v`) wind field, the rotated latitude and longitude points (`rlat` and `rlon`), the latitude and the longitude of the rotated north pole (`pollat`, `pollon`) and the dimensions of the wind fields (`idim` and `jdim`).

The additional subroutine `UVROT2UV_VEC_JLOOP` allows for the transformation of 2D wind fields covering one horizontal and the vertical dimension. This subroutine was implemented as in some codes (e.g. ECHAM5) a so-called "local loop" is used in which only one horizontal dimension is accessible.

3.4 MESSY_MAIN_GRID_TOOLS

This module contains routines for the conversion of data between the multi-dimensional array representation models and the 1D vector representation of the GRID SMCL routines.

3.4.1 RGTOOL_CONVERT

This subroutine converts a variable (`var`) of type `t_narray`, which is defined on the geohybgrid `grid` to a 4D array (`dat`).

```
!-------------------------------------------------------------------------------
  SUBROUTINE RGTOOL_CONVERT(var, dat, grid, order)

    ! CONVERTS NCREGRID OUTPUT OF TYPE N-ARRAY (var) ON GRID
    ! grid TO 4D-ARRAY (dat)
    ! THE OPTIONAL STRING order DEFINES THE ORDER OF DIMENSIONS
    ! (DEFAULT: 'xyzn')
    !
    ! Author: Patrick Joeckel, MPIC, October 2002

    USE ...

    ! REGRID MODULES

    IMPLICIT NONE

    INTRINSIC :: ASSOCIATED, INDEX, PRESENT, PRODUCT, REAL, SIZE, SUM, TRIM

    ! I/O
    TYPE (t_ncvar),    INTENT(IN)           :: var   ! nc-variable
    REAL(dp), DIMENSION(:,:,:,:), POINTER   :: dat   ! DATA ON DESTINATION GRID
    TYPE(t_geohybgrid),INTENT(IN), OPTIONAL :: grid  ! grid information
    CHARACTER(LEN=4),  INTENT(IN), OPTIONAL :: order ! DEFAULT: 'xyzn'
!-------------------------------------------------------------------------------
```

The order of the dimensions of the 4D array is determined by the optional string `order`. If `order` is not present, the default 'xyzn' is assumed.

Internally, the subroutine

1. determines the required order of `dat`. The 1D vector of length 4, variable `ovec`, contains the index of the x-axis or first horizontal axis (`ovec(1)`), of the y- or second horizontal axis (`ovec(2)`), of the vertical axis (`ovec(3)`) and of the number or parameter dimension (`ovec(4)`). The associated strings `'x/LON'`, `'y/LAT'`, `'z/LEV'` and `'n/PAR'` are stored in the respective order in the character vector variable `ostr`.
2. loops over the grid dimensions and compares each dimensions of `grid` with the dimensions of the variable `var`, the respective dimension lengths are stored in the vector variable `ldvar` of length 4 at the same index as the corresponding dimension of `var`. Additionally, the vector variable

`dpos` of length 4 stores the indices of the four dimensions in the order 'x,y,z,n', i.e., `dpos(1)` stores the index of the dimension of grid, which contains the 'x'-axis. Furthermore, the number of identified dimensions is counted (with `nrvdim` as integer counter).

3. the length of the parameter dimension `ldvar(ovec(4))` is determined by

$$ldvar(ovec(4)) = npdlv/(ldvar(ovec(1)) * ldvar(ovec(2)) * ldvar(ovec(3))), \quad (7)$$

   `nldlv` is the size of the overall array. Thus the parameter dimension is determined by dividing the overall size by the product of the 3 spacial dimensions.

4. if the verbosity is high enough, the information about the dimensionality of the variables and the association of dimensions is written to the log-file.

5. Finally, the 4D output array `dat` is allocated to the respective dimension length, i.e.,

```
ALLOCATE DAT(ldvar(1),ldvar(2),ldvar(3),ldvar(4))
```

   and filled: for each entry i of the 1D array of `var`, first the respective indices in the 4D space as defined in grid (`vec`) are determined by calling the function `ELEMENT`. Afterwards `vec` is re-ordered according to the order of x-,y-, z- and n-axis in the 1D (source) and the 4D (destination) array, yielding the index vector `nvec`. Finally, the value of `var%dat` at the current position is copied to `dat` at the positions given by `nvec`: `dat(nvec(1), nvec(2), nvec(3), nvec(4)) = var%dat%vd(i)`

3.4.2 RGTOOL_CONVERT_DAT2VAR

This subroutine is the reverse of the subroutine `RGTOOL_CONVERT`. It converts a 4D array `dat` of order "`order`" to a variable of type `t_narray` on the grid "`GRID`". This subroutine requires as additional input the name of the variable (`vname`) as this needs to be set in the output variable of type `t_narray`.

3.4.3 RGTOOL_G2C

This subroutine converts the components of a grid of type `t_geohybgrid` to "normal" arrays. More precise, it converts the hybrid coefficients (`hyam`, `hybm`, `hyai` and `hybi`), the reference and the surface pressure (`p0` and `ps`) the 1D longitude and latitude components `latm`, `lati`, `lonm` and `loni` to 1D arrays (except for `ps`, which will be converted to a 2D array). This conversion routine does not work for the curvilinear and rotated grid variables.

Internally, this subroutine calls `RGTOOL_CONVERT` for all of the listed grid components.

3.4.4 ALINE_ARRAY

This subroutine converts a 2D array (`var2d`) to a 1D vector (`var1D`). Thus `var1D` and a `status` flag are the output parameters of this subroutine. Input are the 2D array which should be alined into the 1D variable.

3.4.5 DEALINE_ARRAY

This subroutine provides the reverse operation of `ALINE_ARRAY`. It converts a 1D array to a 2D variable.

```
!-------------------------------------------------------------------------------
  SUBROUTINE dealine_array(status, dim1, dim2, var1d, var2d)

    IMPLICIT NONE

    ! I/O
    INTEGER,                  INTENT(OUT) :: status
    INTEGER,                  INTENT(IN)  :: dim1, dim2
    REAL(dp), DIMENSION(:),   INTENT(IN)  :: var1d
    REAL(dp), DIMENSION(:,:), POINTER     :: var2d
!-------------------------------------------------------------------------------
```

A `status` flag and a 2D array `var2d` are the output parameters of this subroutine. Input parameters are the dimensions of the variable `var2d` and the 1D array `var1d`.

3.4.6 SET_SURFACE_PRESSURE

This subroutine (re-)sets the surface pressure component in a geohybrid grid. As, in contrast to the other grid defining parameters, the surface pressure might change during a model simulation, it might become necessary to use the updated surface pressure for vertical remapping in NREGRID model.

Parameters to the subroutine are an integer status flag `status` reporting about success or failure of the subroutine, the grid variable (`grid`) of type `t_geohybgrid` to which the surface pressure field will be copied, the 2D surface pressure field `press` and, optionally, the logical 2D field `lcalc`, which is `.TRUE.` in those columns where vertical remapping takes place. If `lcalc` is not present, it is assumed that vertical remapping will be performed in the entire domain.

3.5 MESSY_MAIN_GRID_TRAFO_NRGD_BASE

This module contains the core of the NCREGRID submodel. It is published and documented by Jöckel (2006). Therefore the contents will not be reported here again. All subroutines now contained in `MESSY_MAIN_GRID_TRAFO_NRGD_BASE` (i.e., `NREGRID`, `NREGRID_STAT` and the subroutines calculating the overlap between region `OVL_*`) are part of the `NCREGRID` SMCL file `messy_ncregrid_base.f90`.

3.6 MESSY_MAIN_GRID_TRAFO_NRGD

This module contains one public, `REGRID_CONTROL`, and four private routines. The four private routines are used within `REGRID_CONTROL`, but are specific for NREGRID requirements (geographically-rectangular grids). Therefore they have been included in the NREGRID module and not, as the other subroutines, in the general transformation module `MESSY_MAIN_GRID_TRAFO`.

3.6.1 REGRID_CONTROL

`REGRID_CONTROL` drives the grid mapping algorithm NREGRID and is called several times within a regrid loop. Internally, `REGRID_CONTROL` is split into two subroutine `REGRID_CONTROL_INIT` and `REGRID_CONTROL_WORK`. `REGRID_CONTROL_INIT` conducts the initialisation of the local variables deduced from the optional arguments and the deallocation of the data fields allocated during the other calls of `REGRID_CONTROL`.

```
! ------------------------------------------------------------------------------
SUBROUTINE REGRID_CONTROL( grid_in, grid_out, tvar, var        &
                                , RG_TYPE, lint                &
                                , lrgx, lrgy, lrgz             &
                                , lfirsto                      &
                                , lpresaxis                    &
                                , lwork                        &
                                , lstatout                     )

  IMPLICIT NONE

  TYPE (t_geohybgrid), INTENT(IN) :: grid_in    ! input  grid info
  TYPE (t_geohybgrid), INTENT(IN) :: grid_out   ! output grid info
  ! list of input  variables
  TYPE (t_ncvar), DIMENSION(:), POINTER    :: tvar
  ! list of output variables
  TYPE (t_ncvar), DIMENSION(:), POINTER    :: var

  INTEGER,  INTENT(IN)            :: RG_TYPE(:) ! regrid type
  LOGICAL,  INTENT(IN)            :: lint       ! input time ?
  LOGICAL,  INTENT(IN) , OPTIONAL :: lrgx       ! regrid along 'lon'
  LOGICAL,  INTENT(IN) , OPTIONAL :: lrgy       ! regrid along 'lat'
  LOGICAL,  INTENT(IN) , OPTIONAL :: lrgz       ! regrid along 'lev'
  LOGICAL,  INTENT(IN) , OPTIONAL :: lfirsto    ! first output step
  LOGICAL,  INTENT(IN),  OPTIONAL :: lpresaxis  ! pressure axis regrid
  LOGICAL,  INTENT(IN),  OPTIONAL :: lwork      ! i_am_worker = T ?
!------------------------------------------------------------------------------
```

While `REGRID_CONTROL_INIT` is, in a distributed memory parallel application called by each task, `REGRID_CONTROL_WORK` does the actual regridding work and is only called on the so-called "worker-PEs", i.e., those PEs, which are dedicated to perform the remapping. If a PE is a worker-PE is indicated by the optional argument `lwork`. If `lwork=.TRUE.`, the PE performs the remapping. If `lwork` is not present, `.TRUE.` is the default.

The other arguments of `REGRID_CONTROL` are

- the source grid (`grid_in`),
- the destination grid (`grid_out`),
- the list of variables to be regridded (`tvar`),
- the list of the finally regridded variables (`var`),
- the regridding type `RG_TYPE`, which is one of `INT`, `EXT`, `IXF` and `IDX`, for regridding intensive or extensive variables, index fraction and index regridding, respectively. The default is `INT`.
- `lint` denoting, if the time information of the input grid or the time information of the output grid should be used for the finally regridded variables.
- `lrgx`, `lrgy` and `lrgz` indicating whether regridding along the x-, y- and z-axis is requested, respectively. If these optional arguments are not present, the default is `.TRUE.`, i.e., regridding along that axis is performed.
- the logical `lfirsto` indicating, whether export of the grid and the variables to a file is performed for the first time. This is important, as attributes are only written once. If `lfisto` is not present, attributes are not written.
- the logical `lpresaxis` signalling whether vertical regridding takes place along the pressure or the sigma coordinate. The default is `.FALSE.` i.e., regridding in sigma coordinates will be performed.

The workflow of `REGRID_CONTROL_WORK` is as follows:

- Preparation of grids and variables for the regridding.
  - Grid operations
    * Allocation of local variables to number of actual variables.
    * Creation of local copies `gi` and `gg` of input and output grid, respectively.
    * Check of input grid `gi` by calling `CHECK_GEOHYBGRID`.
    * If horizontal interpolation is required, switching of output grid `gg` by calling `SWITCH_GEOHYBGRID`, i.e., the grid is reordered such that the regridded axes come first.
    * Check of output grid `gg` by calling `CHECK_GEOHYBGRID`.
    * Balancing of the time axes of `gi` and `gg` by calling `BALANCE_GEOHYBGRID_TIME`, i.e., both grids are adjusted to the same time, i.e., the input grid time if `lint=.TRUE.` the output grid time otherwise.
    * Balancing of surface pressure of `gi` and `gg` by calling `BALANCE_GEOHYBGRID_PS`.
    * If only vertical interpolation is required, switch output grid `gg` by calling `SWITCH_GEOHYBGRID` here.
    * Sorting of input grid by calling `SORT_GEOHYBGRID`. `gis` is the sorted input grid and `gix` the index grid required for the "un-sorting" of variables, i.e., the transformation back to it original order.
    * Sorting of output grid by calling `SORT_GEOHYBGRID`. `ggs` is the sorted output grid and `ggx` the index grid required for the un-sorting of variables.

  - Completion of the sorted and the "index grid" `gis` and `gix` by calling `COMPLETE_GEOHYBGRID`. In this subroutine missing components for interfaces or mid-point variables are calculated from the other component, i.e., interfaces from mid-points or mid-points from interfaces. If both, interfaces and mid-points, are unavailable, these components stay undefined.
  - Completion of the sorted and the "index grid" `ggs` and `ggx` by calling `COMPLETE_GEOHYBGRID`.
  - Balancing the sorted input and output grids `gis` and `ggs` with each other, i.e., missing information in the one grid is added by the respective information of the other grid. Struktur components defined or undefined in both grids stay as they are.
  - Balancing the sorted input and output index grids `gix` and `ggx` with each other.
  - Construction of axis information by calling `GEOHYBGRID_AXES`. `sax` and `dax` contain the axes information for the input and output grid respectively.

- Variable operations:
  In a loop over the variables (indexed `i`) to be regridded the following preparations are made:
  - Check of conformity of variable with input grid `gi` by calling `CHECK_NCVAR_ON_GEOHYBGRID`. If yes, addition of variable to list of regridding variables and copying of `tvar(i)` to local variable `xivar(i)`.
  - If `IXF` or `IDX` regridding is required, call of subroutine `IDX2FRAC_NCVAR` (see Sect. 3.1.2.12). `IDX2FRAC_NCVAR` outputs the new local variable `qvari`. If `IDX2FRAC_NCVAR` is not called, copying of `xivar(i)` to the local variable `qvari`.
  - Check of conformity of variable `qvari` with sorted input grid `gis` definition by calling `CHECK_NCVAR_ON_GEOHYBGRID`.
  - Balancing of the output variable `xovar(i)` with input variable `qvari` and sorted output grid `ggs` by calling `BALANCE_GEOHYBGRID_NCVAR` (see Sect. 3.3.10).
  - Sorting of input variable `qvari` according the input index grid `gix` by calling `SORT_GEOHYBGRID_NCVAR`. The sorted variable is `svari`.
  - Balancing of the sorted output variable `svaro(i)` with input variable `svari` and sorted output grid `ggs` by calling `BALANCE_GEOHYBGRID_NCVAR` (see Sect. 3.3.10).
  - Packing sorted input variable `svari` by calling `PACK_GEOHYBGRID_NCVAR`. The packed variable is called `pvari`.
  - Balancing of the packed output variable `pvaro(i)` with packed input variable `pvari` and sorted output grid `ggs` by calling `BALANCE_GEOHYBGRID_NCVAR` (see Sect. 3.3.10).
  - Deallocation / Initialisation of the data component of `pvaro(i)` by calling `INIT_NARRAY(pvaro(i)%dat)`.
  - Copying of the content of data array of packed input variable `pvari` to variable `nai(i)` of type `t_narray`, i.e., the variable which is input to the NREGRID algorithm.
  - Deallocation / Initialisation of local input variables `qvari`, `svari`, `pvari` by calling `INIT_NCVAR`.

  After this is done for every variable the data is prepared for regridding.

- Regridding:
  The regridding algorithm `NREGRID` is called for all variables at the same time. Parameters to the subroutine are

  - the input and output data arrays `nai` and `nao`, respectively.

  - the input and output (source and destination) data axes information `sax` and `dax`, respectively.
  - the regridding type `RGT` that is requested individually for each variable.
  - the global overlap fractions `sovl` and `dovl` for source and destination grid, respectively.
  - the counter for the recursive level `rcnt`, as NREGRID is a algorithm being called for each axis recursivly.

  If requested, a statistic of the regridding process can be output into the log file afterwards by calling `NREGRID_STAT`.

- Reformatting of output variables and grids:
  After the regridding the data needs to be transformed back to the requested order.

  - In a loop over all regridded variables (loop index `i`) the variables are transformed back into their requested form and order:
    * Copying of the contents of the data array `nao`, output of NREGRID, to the data array of the packed output variable `pvaro(i)` by calling `COPY_NARRAY`.
    * Check of the sorted variable `svaro(i)` on the sorted output grid by calling `CHECK_NCVAR_ON_GEOHYBGRID` to get the definition of output axes (`axes`) and dimensions (`dims`).
    * Unpacking of packed output variable `pvaro(i)` using `axes` and `dims` by calling `PACK_GEOHYBGRID_NCVAR`. Output of this subroutine is the sorted output variable `svaro(i)`.
    * Un-sorting of sorted output variable `svaro(i)` by calling `SORT_GEOHYBGRID_NCVAR` using the output index grid `ggx` and the `axes` definitions. Output of this subroutine is the variable `qvaro`, which is in the original order compared to the input variable `tvar`.
    * For 'IDX' regridded variables, backtransition of the data by calling the subroutine `MAXFRAC2IDX_NCVAR`. Otherwise, copying of `qvaro` to the final output variable `xovar(i)`.
    * Re-initialisation of the local variables `pvaro(i)`, `svaro(i)`, `qvaro`.
  - Memory deallocation:
    The variables `nai`, `nao`, `pvaro` and `svaro` are initialised and deallocated and the corresponding pointers are nullified.
  - The unsorted, completed and balanced output grid `go` is calculated by calling `SORT_GEOHYBGRID`.
  - If a filename is defined in the grid structure `go`, direct output of the regridded fields is requested. First `go` is written to the file by the subroutine `EXPORT_GEOHYBGRID`. Then, if called for the first time (`lfirsto = .TRUE.`), the attributes are written to the file by `EXPORT_NCATT`. Finally, the variables are written to the file using `EXPORT_NCVAR`.
  - Preparation of the returned data: The output variable `var` is allocated to the number of actually regridded variables and the content of the local output variable `xovar` is copied to `var`.

- At the end internally used memory is released.

3.6.2 GEOHYBGRID_AXES

This subroutine constructs the axes from the grid information.

```
!-------------------------------------------------------------------------------
SUBROUTINE GEOHYBGRID_AXES(g, a, g2, a2, pflag)

  ! Note: NO CHECKING
  !       g, g2 must be 'ordered', 'complete', and 'consistent'

  IMPLICIT NONE

  ! I/O
  TYPE (t_geohybgrid), INTENT(IN)      :: g  ! GEOHYBRID-GRID
  TYPE (t_axis), DIMENSION(:), POINTER :: a  ! LIST OF AXES FOR REGRIDDING
  TYPE (t_geohybgrid), INTENT(IN)     , OPTIONAL :: g2  ! GEOHYBRID-GRID
  TYPE (t_axis), DIMENSION(:), POINTER, OPTIONAL :: a2  ! LIST OF AXES
  LOGICAL, OPTIONAL, INTENT(IN)        :: pflag  ! .true.: pressure axis
                                                 ! .false. sigma-axis (default)
!-------------------------------------------------------------------------------
```

Input to the subroutine are one or two geohybgrid grids (`g` and `g2`), and, optionally, a flag, `pflag`, indicating, if a pressure or a sigma axis is used as vertical axis. Output of the subroutine are one or two 1D pointer arrays of axis definitions of type `t_axis` (`a` and `a2`). If a second grid and a second axis array is provided, the procedure is the same as for the first pair, so we explain the procedure here referring only to one grid and one axis array. The axis array is defined according to the grid definition. The following order of actions is taken:

1. The number of axes is determined, by checking if the variables `g%lati`, `g%loni` and `g%hyai` or `g%hybi` are defined. This subroutine assumes, that the input grids are already 'ordered', 'complete' and 'consistent'.

2. The 1D axis pointer array is allocated to the number of dimensions (a free dimension is always assumed to exist in addition).

3. If `g` contains a longitude axis the corresponding definitions are copied from the grid definition and `g%loni%dat` is copied to `a(n)%dat`.

4. The latitude axis is converted from geographical to mathematical coordinates in radians.

5. If a vertical axis exists, the axis is processed depending on the request: by calling the subroutine `H2PSIG` the vertical axis will be transformed to pressure or sigma coordinates. Additionally, the dependence of the vertical axis on the longitude and/or latitude axes is checked.

As this subroutine only checks the non-rotated, non-curvilinear longitude and latitude coordinates, it is only applicable within NREGRID.

3.6.3 PS2PS

This subroutine basically extracts a variable of type `t_narray` from a variable of type `t_ncvar`. As this subroutine only checks the non-rotated, non-curvilinear longitude and latitude coordinates, it is only applicable within NREGRID. As the name of the subroutine indicates, this subroutine is used for extraction of the surface pressure.

3.6.4 BALANCE_GEOHYBGRID_PS

This subroutine adjusts the surface pressure definition of two grids to each other.

First some consistency checks have to be performed:

- In case none of the two grids contains a surface pressure definition, the hybrid-b coefficients must also be undefined.
- In case none of the two grids contains a reference pressure definition, the hybrid-a coefficients must also be undefined.
- The subroutine returns, if the grid is 2D, i.e., neither the surface or the reference pressure, nor any hybrid coefficients are defined.

If the consistency checks are passed, the grids can be balanced. If `p0` is defined in one of the grids and not in the other, the defined reference pressure is copied to the other grid. For the surface pressure itself, four different cases can occur:

A) `ps` is defined in both grids. Thus only the time axis of both grids needs to be adjusted, but this is done in the subroutine `BALANCE_GEOHYBGRID_TIME`.

B) `ps` is undefined in both grids. In this case nothing is to be done.

C) the surface pressure of the outgoing grid `go%ps` is defined, but on the wrong grid.
In this case an error message is produced and the remapping is interrupted.

D) If the incoming or the outgoing surface pressure are defined, the missing surface pressure is remapped by calling the subroutine `REGRID_GEOHYBGRID_PS`.

As `REGRID_GEOHYBGRID_PS` calls NREGRID, this subroutine is only applicable for NREGRID, i.e. non-curvilinear grids.

3.6.5 REGRID_GEOHYBRID_PS

This subroutine basically uses the same algorithm as `REGRID_CONTROL` to horizontally remap the surface pressure defined on one grid to the surface pressure field of another grid.

As this subroutine calls NREGRID, it is only applicable within NREGRID.

3.7 MESSY_MAIN_GRID_TRAFO_SCRP_BASE

This module contains the core of the SCRIP software. Its contents are published by Jones (1999) and on the SCRIP homepage (http://oceans11.lanl.gov/trac/SCRIP (last access data: 05-11-2014)).

Therefore the content of the module is not further described here.

3.8 MESSY_MAIN_GRID_TRAFO_SCRP

This module builds the interface to the original SCRIP interpolation software. As SCRIP uses another type of grid description, this module defines a "SCRIP grid" i.e., the data type `t_scrip_grid` containing the information required by SCRIP for the remapping from or to that grid. In addition to the grid definition itself, there is a concatenated list of SCRIP grids named `SCRIPGRIDLIST`. `SCRIPGRIDLIST` is of type `t_scrip_grid_list`:

```
!-------------------------------------------------------------
 TYPE t_scrip_grid_list
     TYPE(t_scrip_grid)               :: this
     TYPE(t_scrip_grid_list), POINTER :: next => NULL()
 END type t_scrip_grid_list
!-------------------------------------------------------------
```

Using this list, all grids can be stored and searched. If exactly the same definition is required multiple times, no additional grid need to be defined thus saving memory.

In addition to the definition of SCRIP grids, a data type defining a "SCRIP data set" (`t_scrip_data`) is defined. This structure contains the information required for the remapping of one grid to another. Thus, it contains the source and the destination grid, both of type `t_scrip_grid`, and the weights. The weights are typically calculated once during the initialisation phase of the model and stored in a variable of type `t_scrip_weights`.

As for the SCRIP grids, the SCRIP data sets are stored within a concatenated list (`SCRIPDATALIST`). This makes a search for a specific combination of grids and interpolation methods possible in order to avoid calculation of the same weights twice. This saves computing time and memory.

3.8.0.1 t_scrip_grid

The structure `t_scrip_grid` is defined as:

```
!-------------------------------------------------------------------------------
  TYPE t_scrip_grid
     INTEGER :: ID      = -99
     INTEGER :: size    = 0   ! "Horizontal" size product of lon/lat grid dims
                              ! i.e. a 1D size horizontal part  of the grid
     INTEGER :: corners = 4   ! number of corners, in our case always 4
     INTEGER :: rank    = 0   ! number of dimensions in "model code"

     ! length of dimensions in "model code" (dimensioned by rank)
     TYPE (t_ncdim), DIMENSION(:), POINTER :: dim => NULL()

     LOGICAL,        DIMENSION(:), POINTER :: lmask ! dimensioned by grid_size
                                                    ! T for participating points
                                                    ! F for neglected points

     ! longitude of grid centers (dim: grid_size ; unit: radian)
     REAL(dp), DIMENSION(:),   POINTER :: center_lon
     ! latitude of grid centers (dim: grid_size ; unit: radian)
```

```
    REAL(dp), DIMENSION(:),   POINTER :: center_lat
    ! longitude of grid corners (dim: (grid_size, grid_corners); unit: radian)
    REAL(dp), DIMENSION(:,:), POINTER :: corner_lon
    ! latitude of grid corners (dim: (grid_size, grid_corners); unit: radian)
    REAL(dp), DIMENSION(:,:), POINTER :: corner_lat
  END type t_scrip_grid
!------------------------------------------------------------------------------
```

The structure `t_scrip_grid` contains

- an `ID` to unambiguously identify the grid.
- the `size` of the horizontal grid, i.e. number of grid cells in longitude direction times number of cells in latitude direction. In case of 1-dimenisonal grids it is simply the number of grid cells in a horizontal plane.
- `corners`, indicating the number of corners of the grid. As until now only rectangular cases have been implemented, the default is set to 4. Nevertheless, SCRIP is able to deal with different number of corners, but GRID might need some further extension for these grids.
- the `rank` storing the number of spacial dimensions the grid corresponds to.
- a vector `dim`, dimensioned by `rank`, providing the length of the dimension axes of the corresponding spacial grid.
- a logical mask (`lmask`) used by SCRIP to reduce the search area.
- the longitude of each grid cell center (mid point) in radian (`center_lon`).
- the latitude of each grid cell center (mid point) in radian (`center_lat`).
- the longitude of each corner of each grid cell in radian (`corner_lon`). It is a 2D field dimensioned by the number of corners and the number of grid cells.
- the latitude of each corner of each grid cell in radian (`corner_lat`). It is a 2D field dimensioned by the number of corners and the number of grid cells.

3.8.0.2 t_scrip_data

The structure `t_scrip_data` combines the two grids (source and destination grid) and the weights required for the interpolation from the one to the other grid in one data type.

```
!------------------------------------------------------------------------------
  TYPE t_scrip_data
    ! SCRIP data ID
    INTEGER                      :: ID             = 0
    ! remapping type
    INTEGER                      :: map_type       = map_type_conserv
    ! normalize option
    INTEGER                      :: norm_opt       = norm_opt_none
    ! use source grid area
    LOGICAL                      :: luse_sgrd_area = .FALSE.
```

```
    ! use destination grid area
    LOGICAL                      :: luse_dgrd_area = .FALSE.
    TYPE(t_scrip_weights)        :: wghts
    TYPE(t_scrip_grid), POINTER  :: sgrd
    TYPE(t_scrip_grid), POINTER  :: dgrd
  END type t_scrip_data
!-------------------------------------------------------------------------------
```

The source and destination grids (`sgrd` and `dgrd`) are both pointers, as the original grid definition is located in the list of SCRIP grids `SCRIPGRIDLIST`. As the weights are unique for a specific pair of grids and the chosen mapping type, they are stored within this structure. It contains an `ID` for unique identification of a specific SCRIP data set. SCRIP provides different interpolation and normalisation types. As the weights depend on the interpolation type, the `map_type` is a structure component here. Additionally, the normalisation specification is stored in the component `norm_opt`.

3.8.0.3 t_scrip_weights

Internally, SCRIP works with 1D data arrays. Therefore, all non-zero-dimensional components of the type `t_scrip_weights` are 1-dimensional.

```
!-------------------------------------------------------------------------------
 TYPE t_scrip_weights
    ! number of unique address pairs in the remapping == number of entries
    ! in the sparse matrix for the remapping
    INTEGER :: num_links = 0

    ! number of required weights
    !    - bilinear          num_wgts = 1
    !    - distance-weighted num_wgts = 1
    !    - conservative      num_wgts = 3
    !    - bicubic           num_wgts = 4
    INTEGER                       :: num_wgts  = 0

    ! normally the weights are calculated only at the beginning of the
    ! simulation. If ltimedependent is set to true, the weights are recalculated
    ! each import time step.
    ! Note: this is much more computing time and memory intensive and is only
    ! required, if time dependent masks (e.g. ice mask) are important for the
    ! remapping
    LOGICAL                         :: ltimedependent = .FALSE.

    ! remap matrix dimensioned by (num_wghts, num_links)
    REAL(dp), DIMENSION(:), POINTER :: weights => NULL()
    REAL(dp), DIMENSION(:), POINTER :: dstfrac => NULL()  ! fraction
    ! area of destination field
    REAL(dp), DIMENSION(:), POINTER :: dstarea => NULL()
    ! source address of each link (dim: num_links)
    INTEGER,  DIMENSION(:), POINTER :: srcadd  => NULL()
    ! destination address of each link (dim: num_links)
```

```
    INTEGER,  DIMENSION(:), POINTER :: dstadd  => NULL()
  END type t_scrip_weights
!-----------------------------------------------------------------------------
```

The data internally determined by SCRIP during the weight calculation is stored in a variable of type `t_scrip_weights`. The components are:

- integer `num_links`: number of overlapping regions.
- integer `num_wghts`: number of required weights (dependent on interpolation type).
- logical `ltimedependent`: indicating whether the weights are time dependent; if so, the weights need to be calculated each timestep. IMPORT_GRID only handels time independent grids so far.
- 1D float pointer array `weights`: currently only interpolation methods requiring one weight are implemented. Thus `weight` will be dimensioned by `num_links`.
- 1D float pointer array `dstfrac`: fraction of destination grid cells covered by linked source grid cell, dimensioned by `num_links`.
- 1D float pointer array`dstarea`: grid box area of each grid box of the destination grid and thus dimensioned by the number of grid cells of the destination grid.
- 1D float pointer array `srcadd`: source address of each link, thus dimensioned by `num_links`.
- 1D float pointer array `dstadd`: destination address of each link, thus dimensioned by `num_links`.

3.8.1 INIT_SCRIPGRID

The subroutine `INIT_SCRIPGRID` initialises a variable of type `t_scrip_grid`.

- `grid%ID` is initialised with -99.
- `grid%size` is set to 0,
- `grid%corners` to 4 and
- `grid%rank` to 0.
- For `grid%dim` first all dimensions are initialised using the subroutine `INIT_NCDIM`, afterwards, `grid%dim` is deallocated and nullified.
- `grid%lmask`, `grid%center_lon`, `grid%center_lat`, `grid%corner_lon` and `grid%corner_lat` are deallocated and nullified.

3.8.2 COPY_SCRIPGRID

This subroutine copies a source grid `sgrid` of type `t_scrip_grid` to a destination grid `dgrid` of the same type.

3.8.3 DEFINE_SCRIPGRID

The subroutine `DEFINE_SCRIPGRID` defines a variable of type `t_scrip_grid` and adds it to the concatenated list of SCRIP grids.

```
! ============================================================================
  SUBROUTINE define_scripgrid (status, rank, size, corners       &
       , dims, lmask, clon, clat, corlon, corlat, id, grid, pgrid)

    USE messy_main_grid_netcdf, ONLY: INIT_NCDIM

    IMPLICIT NONE

    ! I/O
    INTEGER, INTENT(OUT)           :: status
    INTEGER, INTENT(IN)            :: rank
    INTEGER, INTENT(IN)            :: size
    INTEGER, INTENT(IN)            :: corners
    ! FIELD in
    INTEGER,  DIMENSION(:),   INTENT(IN) :: dims
    LOGICAL,  DIMENSION(:),   INTENT(IN) :: lmask
    REAL(dp), DIMENSION(:),   INTENT(IN) :: clon
    REAL(dp), DIMENSION(:),   INTENT(IN) :: clat
    REAL(dp), DIMENSION(:,:), INTENT(IN) :: corlon
    REAL(dp), DIMENSION(:,:), INTENT(IN) :: corlat

    INTEGER,             INTENT(  OUT), OPTIONAL :: ID
    TYPE(t_scrip_grid), INTENT(INOUT), OPTIONAL :: grid
    TYPE(t_scrip_grid), POINTER,       OPTIONAL :: pgrid
! ============================================================================
```

Input to this subroutine are the the `rank`, the `size`, the number of `corners`, the number of dimensions (`dim`), the logical mask (`lmask`), the center longitudes and latitudes (`clon`, `clat`) and the corner longitudes and latitudes (`corlon`, `corlat`). Output of the subroutine are a mandatory `status` flag, reporting failure or success of the subroutine, the `ID` of the newly added grid, in the concatenated list of SCRIP grids, a variable `grid` of type `t_scrip_grid` containing all data given to the subroutine, and a pointer to the respective SCRIP grid in the concatenated list `SCRIPGRIDLIST`.

Internally, the subroutine cycles the concatenated list of SCRIP grids `SCRIPGRIDLIST` and compares each grid of this list to the actual parameters of the subroutine. This comparison is done by the subroutine `COMPARE_TO_SCRIPGRID` (Sect. 3.8.5). If an identical grid is found, the optional output parameters are filled with the corresponding grid information. If no identical grid is found, an additional entry in `SCRIPGRIDLIST` is created using all input parameters. At the end, the optional output parameters of the subroutine are filled, if present.

3.8.4 CLEAN_SCRIPGRID_LIST

This subroutine deletes the concatenated list of SCRIP grids (`SCRIPGRIDLIST`). Internally, the subroutine cycles through all list entries. First a pointer to the next list element is stored. Afterwards,

the current list entry is (re-)initialised by calling `INIT_SCRIPGRID`. Finally, the pointer to the current list entry is deallocated and nullified. Using the pointer to the next list element, the deletion process is carried forward until all list elements are erased.

3.8.5 COMPARE_TO_SCRIPGRID

The subroutine `COMPARE_TO_SCRIPGRID` compares a list of variables corresponding to the components of a structure variable of type `t_scrip_grid` to a grid, which is also provided to the subroutine.

```
! ============================================================================
    SUBROUTINE COMPARE_TO_SCRIPGRID(id, grid, rank, size, corners &
             , dims, lmask, clon, clat, corlon, corlat)

    IMPLICIT NONE

    ! I/O
    INTEGER, INTENT(OUT)             :: id
    TYPE(t_scrip_grid), POINTER      :: grid
    INTEGER, INTENT(IN)              :: rank
    INTEGER, INTENT(IN)              :: size
    INTEGER, INTENT(IN)              :: corners
    ! FIELD in
    INTEGER,  DIMENSION(rank),         INTENT(IN) :: dims
    LOGICAL,  DIMENSION(size),         INTENT(IN) :: lmask
    REAL(dp), DIMENSION(size),         INTENT(IN) :: clon
    REAL(dp), DIMENSION(size),         INTENT(IN) :: clat
    REAL(dp), DIMENSION(corners,size), INTENT(IN) :: corlon
    REAL(dp), DIMENSION(corners,size), INTENT(IN) :: corlat
! ============================================================================
```

Input arguments of the subroutine are

- a pointer to the `grid` of type `t_scrip_grid` to which the variables should be compared,
- an integer naming the `rank` of the horizontal grid,
- an integer giving the `size`, i.e. the number of grid cells, of the grid,
- an integer `corners` giving the number of corners of a grid cell,
- an integer 1D vector naming the length of each individual rank of the grid `dims`,
- a logical 1D vector `lmask` giving the logical mask for interpolation,
- a 1D vector `clon` containing the longitudes of the grid cell centers,
- a 1D vector `clat` listing the latitudes of the grid cell centers,
- a 2D array `corlon` providing the longitudes of all corners of each grid cell, and
- a 2D array `corlat` listing the latitudes of all corners of each grid cell.

The only output parameter is an `ID`. At the beginning of the subroutine `ID` is set to -99. If the grid components and the separate variables are equivalent, `ID` is set to the ID of `grid`. Otherwise, an `ID` value of -99 is returned by the subroutine indicating that the `grid` components and the separate variables differ.

3.8.6 GEOHYB2SCRIPGRID

The subroutine `geohyb2scripgrid` converts a variable of type `t_geohybgrid` into a variable of type `t_scrip_grid`.

```
!-------------------------------------------------------------------------------
  SUBROUTINE geohyb2scripgrid(status, ggrid, sgrid, psgrid, ID, l_set_ranges)

    USE ...

    IMPLICIT NONE

    ! I/O
    INTEGER,             INTENT(OUT)             :: status
    TYPE(t_geohybgrid), INTENT(IN)               :: ggrid
    TYPE(t_scrip_grid), OPTIONAL, INTENT(INOUT) :: sgrid
    TYPE(t_scrip_grid), OPTIONAL, POINTER       :: psgrid
    INTEGER,             OPTIONAL, INTENT(OUT)   :: ID
    LOGICAL,             OPTIONAL, INTENT(IN)    :: l_set_ranges
!-------------------------------------------------------------------------------
```

Parameters of this subroutine are

- an integer `status` flag reporting back a possible error,
- the geohybrid grid `ggrid` to be converted,
- the output grid `sgrid` of type `t_scrip_grid`,
- a pointer to the output grid `sgrid` `psgrid`,
- the `ID` of the grid in the concatenated list of SCRIP grids as an optional output argument, and
- an optional logical flag `l_set_ranges` forcing the subroutine to set limited longitude ranges dependent on the grid definition.

The subroutine itself is split into different parts:

1. Initialisation of local variables:
   To simplify the further processing of the data, the local variables `llonm`, `llatm`, `lloni` and `llati` are filled depending on the definitions of the input geohybrid grid. If the grid components `lonm`, `latm`, `loni` and `lati` are defined, these components are copied to the local variables. If they are not defined, the curvilinear components are copied. To indicate, which structure components are assigned to the local variables, the logical `l_curvilinear` is set `.TRUE.`, if only the curvilinear

components are used. Additionally, the two logicals `l_mids` and `l_interfaces` indicate, if the mid-point and the interface components are defined, respectively. To further simplify the processing, the local longitude and latitude variables are converted to double precision, if not already available as such. Last but not least, if the optional subroutine argument `l_set_ranges` is present and `.TRUE.`, the subroutine `set_ranges` checks whether the longitudes are all defined within the intervall `[-180,360]`. If the maximum longitude is smaller than `180` the local range variable `lon_ranges` is set to the interval `[-180,180]`, otherwise the range is set to `[0,360]`. If `l_set_ranges` is not present or `.FALSE.`., `lon_ranges` is set to the interval `[0,360]`.

2. Determination of the grid size:
   The grid size `gsize` is defined as product of the length of the two horizontal dimensions `locdims(1:2)`. The latter are determined according to the available information:

   - If the mid-point variables on a non-curvilinear grid are defined, the locdims are simply the lengths of the longitude and the latitude axes, respectively.
   - If only the interface variables on the non-curvilinear grid are defined, the corresponding local dimensions are the lengths of the axes minus 1.
   - For a curvilinear grid the longitude variable corresponds to a 2D spacial array. Therefore the local dimensions correspond to the length of the first and the second dimension axis of the longitude variable for the mid-point variables. The length of the interface dimensions is larger by one.

3. Determination of the `rank` and `lmask`:
   A `rank` of 2 is assumed for all calculations within this subroutine. If another grid, i.e., a 1D horizontal grid shall be transformed, another transformation routine is required. Finally, a logical map `lmask`, indicating where the interpolation algorithm should search for overlap, is set `.TRUE.` everywhere. This might be improved in the future.

4. Determination of the center and corner longitude and latitudes:
   For SCRIP the longitude and latitudes have to be defined in *radian*. Thus a conversion factor `fac` is defined. Depending on the unit of longitudes in the geohybrid grid `fac` is 1, if the unit is already *radian*. Otherwise, it is assumed that the unit of the longitudes and latitudes is *degrees* and the conversion factor is set to $DTR = pi/180.\_dp$ as defined in the generic MESSy submodel CONSTANTS (file: `messy_main_constants_mem.f90`).

   - Determination of center variables `clon` and `clat`:
     If the **mid-point variables** are defined, the center variables can be simply set. For the **geographically-rectangular grid** the 1D variables `clon` and `clat` are set in loops over the two horizontal dimensions:

```
!----------------------------------------------------------------------
DO i = 1, locdims(1)
   DO j = 1, locdims(2)
     n = (j-1) * locdims(1) + i
     clon(n) = llonm%vd(i) * fac
     clat(n) = llatm%vd(j) * fac
   END DO
END DO
!----------------------------------------------------------------------
```

For a **curvilinear grid**, the longitudes and latitudes are already defined by a 1D vector. Thus `clon` and `clat` are simply set by

```
!------------------------------------------------------------------------
DO n = 1, gsize
   clon(n) = llonm%vd(n) * fac
   clat(n) = llatm%vd(n) * fac
END DO
!------------------------------------------------------------------------
```

If only the **interface variables** are defined, the center variables need to be calculated. For the **geographically-rectangular grid** the centers are in the mid between the interfaces thus the centers are determined by:

```
!------------------------------------------------------------------------
DO i = 1, locdims(1)
   DO j = 1, locdims(2)
      n = (j-1) * locdims(1) + i
      clon(n) = (lloni%vd(i)+lloni%vd(i+1))/2. * fac
      clat(n) = (llati%vd(j)+llati%vd(j+1))/2. * fac
   END DO
END DO
!------------------------------------------------------------------------
```

There is no way to unambiguously determine the centers in geographical longitudes and latitudes for **a curvilinear grid only from the interfaces**. Thus an error message stops the execution of the simulation, if the geohybrid definition is incomplete.

- Determination of corner variables `corlon` and `corlat`:
  For a calculation of the corners it has to be taken into account that the corners must be defined counter-clockwise for the SCRIP algorithm.
  - For a **geographically-rectangular grid** the corners are easily determined from the **interface variables**. To simplify the conversion to counter-clockwisely ordered corners, the helper indices `jll` and `jur` adjust the indices to the correct order, depending on whether the latitude axis is increasing or decreasing.

    ```
    !------------------------------------------------------------------------
    IF (llati%vd(1) <  llati%vd(2)) THEN
       jll = 0
       jur = 1
    ELSE
       jll = 1
       jur = 0
    END IF
    DO j = 1, locdims(2)
       DO i = 1, locdims(1)
           n = (j-1) * locdims(1) + i
           ! counter clockwise start lower left
           ! lower left corner
           tcorlon(1,n) = lloni%vd(i)
           tcorlat(1,n) = llati%vd(j+jll)
    ```

```
      ! lower right corner
      tcorlon(2,n) = lloni%vd(i+1)
      tcorlat(2,n) = tcorlat(1,n)
      ! upper right corner
      tcorlon(3,n) = tcorlon(2,n)
      tcorlat(3,n) = llati%vd(j+jur)
      ! upper left corner
      tcorlon(4,n) = tcorlon(1,n)
      tcorlat(4,n) = tcorlat(3,n)
   END DO
END DO
!-----------------------------------------------------------------------
```

If only the **mid-point variables** are defined for a **geographically-rectangular grid**, the corners are calculated from the mid-points. First, the grid distances `dlon` and `dlat` are determined. Afterwards these are used to calculated the longitudes and latitudes of the corners:

```
!-----------------------------------------------------------------------
 dlon = ABS(llonm%vd(2) - llonm%vd(1))
 DO j = 1, locdims(2)
    IF (j == 1) THEN
       dlat1 = ABS(llatm%vd(2) - llatm%vd(1))
    ELSE
       dlat1 = ABS(llatm%vd(j) - llatm%vd(j-1))
    ENDIF
    IF (j == locdims(2)) THEN
       dlat2 = ABS(llatm%vd(j) - llatm%vd(j-1))
    ELSE
       dlat2 = ABS(llatm%vd(j+1) - llatm%vd(j))
    ENDIF
    DO i = 1, locdims(1)
       n = (j-1) * locdims(1) + i
       ! lower left corner
       tcorlon(1,n) = (llonm%vd(i) - 0.5_dp * dlon)
       tcorlat(1,n) = (llatm%vd(j) - 0.5_dp * dlat1)
       ! lower right corner
       tcorlon(2,n) = (llonm%vd(i) + 0.5_dp * dlon)
       tcorlat(2,n) = tcorlat(1,n)
       ! upper right corner
       tcorlon(3,n) = tcorlon(2,n)
       tcorlat(3,n) = llatm%vd(j) + 0.5_dp * dlat2
       ! upper left corner
       tcorlon(4,n) = tcorlon(1,n)
       tcorlat(4,n) = tcorlat(3,n)
       jur = 10
       if_rgempty: IF (gi%ranges(2,1) /= RGEMPTY .AND. &
            gi%ranges(2,2) /= RGEMPTY) THEN
          IF (j == 1) THEN
             IF (llatm%vd(1) < llatm%vd(2)) THEN
```

```
               tcorlat(1,n) = MINVAL(gi%ranges(2,:))
               tcorlat(2,n) = MINVAL(gi%ranges(2,:))
               jur = 5
            ELSE
               tcorlat(3,n) = MAXVAL(gi%ranges(2,:))
               tcorlat(4,n) = MAXVAL(gi%ranges(2,:))
               jur = 6
            END IF
         ELSE IF (j == locdims(2)) THEN
            IF (llatm%vd(1) < llatm%vd(2)) THEN
               tcorlat(3,n) = MAXVAL(gi%ranges(2,:))
               tcorlat(4,n) = MAXVAL(gi%ranges(2,:))
               jur = 7
            ELSE
               tcorlat(1,n) = MINVAL(gi%ranges(2,:))
               tcorlat(2,n) = MINVAL(gi%ranges(2,:))
               jur = 8
            END IF
         END IF
      ENDIF if_rgempty
   END DO
 END DO
!------------------------------------------------------------------------
```

Finally, if the component `ranges` of the geohybrid grid is set, the longitudes and latitudes of the corners are adjusted accordingly.

– For **curvilinear grid**s calculation of the corners is only possible if the interface variables are defined. If only **mid-point variables** are defined, the simulation is interrupted. For the assignment of the **interface variables** to the corners, the respective indices are calculated using the subroutine `ELEMENT`.

```
!------------------------------------------------------------------------
          ! ... a) interfaces are provided:
          vdim(1) = gi%cloni%dim(1)%len - 1
          vdim(2) = gi%cloni%dim(2)%len - 1

          DO n = 1, SIZE(tcorlon,2)
             ! get element vector for mids
             CALL ELEMENT(vdim,n,ivec)
             ! calculate position in interface array
             iul = n  + vdim(1) + 1 + ivec(2) - 1
             iur = n  + vdim(1) + 1 + ivec(2)
             ilr = n                + ivec(2)
             ill = n                + ivec(2) - 1

             IF (llati%vd(ill) > llati%vd(iul)) THEN
                ! switch upper and lower
                ! enforce coutner-clockwise
                ilr = iul
                iul = n  + ivec(2)
```

```
                ENDIF
                ! lower left corner
                tcorlon(1,n) = lloni%vd(ill)
                tcorlat(1,n) = llati%vd(ill)
                   ! lower right corner
                tcorlon(2,n) = lloni%vd(ilr)
                tcorlat(2,n) = llati%vd(ilr)
                ! upper right corner
                tcorlon(3,n) = lloni%vd(iur)
                tcorlat(3,n) = llati%vd(iur)
                ! upper left corner
                tcorlon(4,n) = lloni%vd(iul)
                tcorlat(4,n) = llati%vd(iul)

                DEALLOCATE(ivec, STAT=status)
                NULLIFY(ivec)
            END DO
        !-------------------------------------------------------------------------
```

So far only the local variables `tcorlon` and `tcorlat` have been calculated. These are now additionally adjusted to the longitude range given by `lon_ranges`, converted to *radian* and finally adjusted to the latitude interval `[-pi/2, pi/2]` as required by SCRIP.

5. Definition of SCRIP grid:
   After the components of a SCRIP grid have been calculated individually, the subroutine `DEFINE_SCRIPGRID` (Sect. 3.8.3) is called to define a variable of type `t_scrip_grid` and to add the grid to the list.

6. Copying of `INTENT(OUT)` variables:
   Depending on their presence, the variable `sgrid` containing the newly defined SCRIP grid, the `ID` containing the ID of the SCRIP grid in the concatenated list of SCRIP grids, and the pointer to the SCRIP grid in the concatenated list `psgrid` are assigned.

7. Clean up:
   Local variables are initialised and deallocated.

3.8.7 INIT_SCRIPDATA

The subroutine `INIT_SCRIPDATA` initialises a variable of type `t_scrip_data`, which is the only argument of this subroutine. The components of the input/output variable `sdata` are initialised as follows:

- `sdata%ID` is set to 0.
- `sdata%map_type` is set to `map_type_conserv`.
- `sdata%norm_opt` is set to `norm_opt_none`.
- `sdata%luse_sgrd_area` is set `.FALSE.` .
- `sdata%luse_dgrd_area` is set `.FALSE.` .

- The weights (`sdata%wghts`) are initialised by calling `INIT_SCRIP_WEIGHTS` (Sect. 3.8.13).
- `sdata%sgrd` and `sdata%dgrd` are initialised by calling `INIT_SCRIPGRID` (Sect. 3.8.1).

3.8.8 DEFINE_SCRIPDATA

The subroutine `DEFINE_SCRIPDATA` defines a variable of type `t_scrip_data` and adds it to the concatenated list of SCRIP data (`SCRIPDATALIST`).

```
!-------------------------------------------------------------------------------
    SUBROUTINE define_scripdata(status, SDAT_id, maptype, normopt &
         , luse_area, dgrd, sgrd, garea2, PSD)

      IMPLICIT NONE

      ! I/O
      INTEGER,          INTENT(OUT) :: status
      INTEGER,          INTENT(OUT) :: SDAT_ID
      CHARACTER(LEN=*), INTENT(IN)  :: name
      INTEGER,          INTENT(IN)  :: maptype
      INTEGER,          INTENT(IN)  :: normopt
      LOGICAL,          INTENT(IN)  :: luse_area

      TYPE(t_scrip_grid),  POINTER  :: dgrd
      TYPE(t_scrip_grid),  POINTER  :: sgrd

      REAL(dp),      DIMENSION(:), POINTER  :: garea2
      TYPE(t_scrip_data), POINTER, OPTIONAL :: PSD

!-------------------------------------------------------------------------------
```

Input parameters are the variables which together determine the components of the newly defined SCRIP data set, i.e.,

- the `maptype`,
- the `normopt`,
- a switch `luse_area` indicating whether a predefined area of the grid shall be used,
- the source SCRIP grid `sgrd` and
- the destination SCRIP grid `dgrd` and
- a pointer which, if associated, points to a 1D array containing the area of each grid cell (`garea2`).

Output parameters are

- the `status` flag,
- the ID (`SDAT_ID`) of and

- the pointer `PSD` pointing to the newly defined SCRIP data set in the concatenated list.

Internally the subroutine cycles through the list of SCRIP data sets and compares each of the already defined data sets with the new components by calling the subroutine `COMPARE_SCRIPDATA` (Sect. 3.8.9). If an equivalent data set is found, `SDAT_id` is set to the equivalent SCRIP data set and, if present, the pointer `PSD` is set to the already existing data set in the `SCRIPDATALIST` before returning from the subroutine.

If none of the SCRIP data sets in the list fits all new components, a new SCRIP data set is added to the concatenated list.

3.8.9 COMPARE_SCRIPDATA

Input parameters are the individual components of a SCRIP data set and a pointer to the SCRIP data set (`data`) to compare the variables to. The source and the destination grid are unambiguously defined by their SCRIP grid ID. The only output parameter is an `ID`. `ID` is set to -99 at the beginning of the subroutine. If components differ, the subroutine returns and an `ID` of -99 indicates that the data sets differ. Otherwise, if all components are equivalent, `ID` is set to the id of the data set (`data%ID`).

3.8.10 LOCATE_SCRIPDATA

This subroutine locates a SCRIP data set in the concatenated list of SCRIP data sets (`SCRIPDATALIST`). Input to this subroutine is the `ID` of a SCRIP data set. Output parameters are a `status` flag and a pointer (`pdata`) which the subroutine associates to the requested SCRIP data set.

Internally, the `SCRIPDATALIST` is searched for a data set with the respective `ID`. If such a data set is found, `pdata` is associated to this data set. Otherwise, `status` is set to error number 3013 indicating that such a data set does not exist.

3.8.11 CLEAN_SCRIPDATA_LIST

This subroutine deletes the complete concatenated list of SCRIP data sets (`SCRIPDATALIST`). Internally the subroutine cycles through all list entries. First the pointer to the next list element is stored. Afterwards, the current list entry is initialised by calling `INIT_SCRIPDATA`. Finally, the pointer to the current list entry is deallocated and nullified.

Subsequently, the next list element is processed in the same way. This continues until all list elements are deleted.

3.8.12 CALC_SCRIPDATA

This subroutine defines a SCRIP data set. It outputs, apart from a `status` flag, an ID (`SCRIP_ID`) of and, optionally, a pointer (`PSD`) to the newly defined SCRIP data set.

```
!-------------------------------------------------------------------------------
  SUBROUTINE calc_scrip_data(status, igrid, ogrid, RGT, SCRIP_ID, oarea,PSD &
       , norm_opt_in, map_type_in)

    USE ...
```

```
    IMPLICIT NONE

    ! I/O
    INTEGER,             INTENT(OUT) :: STATUS
    TYPE(t_geohybgrid), INTENT(IN)  :: igrid
    TYPE(t_geohybgrid), INTENT(IN)  :: ogrid
    INTEGER,             INTENT(IN)  :: RGT(:)  ! regridding type

    INTEGER,             INTENT(OUT) :: SCRIP_ID
    ! OPTIONAL
    REAL(dp), DIMENSION(:,:), POINTER, OPTIONAL :: oarea
    TYPE(t_scrip_data),       POINTER, OPTIONAL :: PSD
    INTEGER,                           OPTIONAL :: norm_opt_in
    INTEGER,                           OPTIONAL :: map_type_in
!------------------------------------------------------------------------------
```

Input parameters are the input and the output geohybrid grids (`igrid` and `ogrid`), the regridding type `RGT` and, optionally, the area of the output grid (`oarea`), the map type (`map_type_in`), and the normalisation option (`norm_opt_in`).

Internally, the following steps are processed:

1. Calculation of the SCRIP input and output grids from the geohybrid grids by calling the subroutine `GEOHYB2SCRIPGRID`.
2. Determination of the `map_type` and `norm_opt`
   Per default, `map_type` and `norm_opt` are determined from the regridding type `RGT`.
   - If `RGT` equals one of `RG_INT`, `RG_IDX` or `RG_IXF`, `norm_opt` is set to `norm_opt_frcarea` and `map_type` is set to `map_type_conserv`.
   - For `RGT==RGT_EXT` `norm_opt` is set to `norm_opt_none` and `map_type` to `map_type_conserv`.
   - However, if required these values can be overwritten by the optional parameters `map_type_in` and `norm_opt_in`, respectively.
3. Transformation of the output gridarea (input parameter `garea2` of `DEFINE_SCRIPDATA`) into a 1D array by calling `ALINE_ARRAY` and conversion to the unit required by SCRIP, if the pointer `oarea` is present and associated. Additionally, `luse_area` is set `.TRUE.` in this case.
4. Creation of a new SCRIP data set by calling `DEFINE_SCRIPDATA`.
5. Re-initialisation of local variables.

3.8.13 INIT_WEIGHTS

This subroutine initialises, deallocates and nullifies (where apropriate) the components of the structure variable `t_scrip_weights`.

3.8.14 CALC_SCRIP_WEIGHTS

This subroutine calculates the weights for the remapping between the two grids and the interpolation methods defined by a SCRIP data set. Input to the subroutine is a pointer to a SCRIP data set (`PSD`), output is a `status` flag, informing about success or failure of the subroutine. The subroutine is divided into three parts:

1. Definition phase:
   The internal SCRIP variables (e.g. `grid1_center_lon`) are set to there counterparts in the SCRIP data set. The SCRIP variables `north_thresh` and `south_thresh` are calculated from the corner latitudes:

   ```
   north_thresh = MAX(MAXVAL(grid1_corner_lat),MAXVAL(grid2_corner_lat))
   south_thresh = MIN(MINVAL(grid1_corner_lat),MINVAL(grid2_corner_lat))
   ```

   Additionally, the variable `babystep` needs to be set. This is a small expansion we introduced to SCRIP: Normally, in the subroutine `intersection` the search step `s1` is increased by `0.001` in each iteration. However, dependent on the grids, a smaller increase might be necessary. As this is very expensive (in terms of computing time), we defined an additional variable called `babystep` which by default is `0.001`, but can be decreased for certain grids. This `babystep` also needs to be set in the definition phase.

2. Calculation phase:
   In this phase the sequence of SCRIP routines is called:

   - `remap_init`
   - `bounds_calc`
   - `remap_vars(1)`
   - `remap_XXX` with `XXX` equals one of `conserv`, `bilin`, `bicub` or `distwgt`, dependent on the `map_type`.
   - `remap_vars(2)`.

3. Saving phase:
   In this phase the results of the interpolation need to be saved in `PSD%wghts`. To be more precise,

   - `num_links_map1` is copied to `PSD%wghts%num_links`,
   - `num_wts` to `PSD%wghts%num_wgts`,
   - `wts_map1(1,:)` to `PSD%wghts%weights`,
   - `grid1_add_map1` to `PSD%wghts%srcadd`,
   - `grid2_add_map1` to `PSD%wghts%dstadd`,
   - `grid2_frac` to `PSD%wghts%dstfrac` and
   - `grid2_area` to `PSD%wghts%dstarea`.

4. Cleaning phase:
   At the end the local variables and pointers need to be deallocated and, where appropriate, nullified. Additonally, the SCRIP internal subroutine `remap_dealloc` is called.

3.8.15 APPLY_SCRIP_WEIGHTS

This subroutine performs the actual interpolation.

```
!-------------------------------------------------------------------------------
  SUBROUTINE APPLY_SCRIP_WEIGHTS(status, pvari, pvaro, PSDID)

    USE  ...

    IMPLICIT NONE

    ! I/O
    INTEGER, INTENT(OUT)                 :: status
    ! INTENT(IN)
    TYPE(t_ncvar), DIMENSION(:), POINTER :: pvari  ! variable to be interpolated
    INTEGER, INTENT(IN)                  :: PSDID
    ! INTENT(INOUT)
    TYPE(t_ncvar), DIMENSION(:), POINTER :: pvaro  ! interpolated fields
!-------------------------------------------------------------------------------
```

Input are the 1D pointer array `pvari` of type `t_ncvar`, which contains the data to be interpolated and the ID of the SCRIP data set `PSDID` comprising the interpolation weights. Output are the interpolated data `pvaro`, i.e., a 1D pointer array of type `t_ncvar` and a `status` flag.

- At the beginning of the subroutine the SCRIP data set is located by calling `LOCATE_SCRIPDATA` (Sect. 3.8.10).
- Even if SCRIP can only be used for horizonal interpolation of the data, the data itself can be of more dimensions. The additional dimensions are invariant dimensions, they are not altered by the horizontal interpolation. The size of this invariant dimension (`nfree`) is determined first: as the source grid size (`PSD%sgrd%size`) contains the product of the horizontal dimensions, `nfree` is simply the quotient of the overall SIZE of the data array and the grid size:

  ```
  nfree = SIZE(pvari(i)%dat%vr) / PSD%sgrd%size  .
  ```

- Knowing the number of free dimensions and the grid size of the horizontal destination grid, the `dat` component of the output variable `pvaro` can be allocated to the correct size:

  ```
   ALLOCATE(pvaro(i)%dat%vr(nfree * PSD%dgrd%size))
  ```

  The length of the pointer array `pvaro` has to be determined outside of `APPLY_SCRIP_WEIGHTS`. Moreover, the above expressions are shown for the single precision variables, but the same statements exist for double precision in the code. This is also the case for the following item.

- Application of the precalculated weights, i.e., the interpolation:
  For the application of the weights, the equations given in the SCRIP user guide for the different normalisation options are applied for each variable:

```
! ------------------------------------------------------------------------
 do_free: DO ifree = 0, nfree-1
    ! shift for each free dimension
    dshft = ifree * PSD%dgrd%size
    sshft = ifree * PSD%sgrd%size
    SELECT CASE(PSD%norm_opt)
    CASE(norm_opt_frcarea)
       IF (pvari(i)%dat%type == VTYPE_REAL) THEN
          DO j = 1, PSD%wghts%num_links
            pvaro(i)%dat%vr(dshft+PSD%wghts%dstadd(j)) = &
                  pvaro(i)%dat%vr(dshft+PSD%wghts%dstadd(j))  &
                  + PSD%wghts%weights(j) * &
                  pvari(i)%dat%vr(sshft+PSD%wghts%srcadd(j))
           END DO
       ELSE IF (pvari(i)%dat%type == VTYPE_DOUBLE) THEN
          ...
       ENDIF
    CASE(norm_opt_dstarea)
       IF (pvari(i)%dat%type == VTYPE_REAL) THEN
          DO j = 1, PSD%wghts%num_links
             IF (PSD%wghts%dstfrac(PSD%wghts%dstadd(j)) /= zero) THEN
                pvaro(i)%dat%vr(dshft+PSD%wghts%dstadd(j)) = &
                     pvaro(i)%dat%vr(dshft+PSD%wghts%dstadd(j))  &
                     + ( PSD%wghts%weights(j) * &
                     pvari(i)%dat%vr(sshft+PSD%wghts%srcadd(j))) &
                     / PSD%wghts%dstfrac(PSD%wghts%dstadd(j))
             ELSE
                pvaro(i)%dat%vr(dshft+PSD%wghts%dstadd(j)) = 0.
             ENDIF
          END DO
       ELSE IF (pvari(i)%dat%type == VTYPE_DOUBLE) THEN
           ...
       ENDIF
    CASE(norm_opt_none)
       IF (pvari(i)%dat%type == VTYPE_REAL) THEN
          DO j = 1, PSD%wghts%num_links
             IF (PSD%wghts%dstfrac(PSD%wghts%dstadd(j)) /= zero) THEN

                pvaro(i)%dat%vr(dshft+PSD%wghts%dstadd(j)) = &
                     pvaro(i)%dat%vr(dshft+PSD%wghts%dstadd(j))  &
                     + (PSD%wghts%weights(j) * &
                     pvari(i)%dat%vr(sshft+PSD%wghts%srcadd(j))) &
                     / (PSD%wghts%dstfrac(PSD%wghts%dstadd(j)) &
                     *PSD%wghts%dstarea(PSD%wghts%dstadd(j)))
             ELSE
                pvaro(i)%dat%vr(dshft+PSD%wghts%dstadd(j)) = 0.
             ENDIF
          END DO
```

```
          ELSE IF (pvari(i)%dat%type == VTYPE_DOUBLE) THEN
             ...
          ENDIF
        CASE DEFAULT
          ! normalize option not implemented
          status = 3030
          RETURN
        END SELECT
      END DO do_free
    ! ------------------------------------------------------------------------
```

For readability the double precision sections have been skipped, but they are in the code. According to the number of invariant dimensions (`nfree`) data "slices" are interpolated in this routine for each variable. For the correct addressing of the individual slices, the shift variables `sshft` and `dshft` for the index shift in the source and in the destination grid are defined.

3.8.16 SCRIP_CONTROL

The subroutine `SCRIP_CONTROL` is the SCRIP counterpart to `REGRID_CONTROL`. Thus the overall procedure is the same.

```
! ------------------------------------------------------------------------------
  SUBROUTINE SCRIP_CONTROL (status, SCRIP_ID, igrid, ogrid, RG_TYPE, lint &
                          , invar, var, grid, llrgz, lfirsto)

   IMPLICIT NONE

    INTEGER, INTENT(OUT)                     :: status
    INTEGER, INTENT(IN)                      :: SCRIP_ID
    TYPE(t_geohybgrid), INTENT(IN)           :: igrid
    TYPE(t_geohybgrid), INTENT(IN)           :: ogrid

    INTEGER, INTENT(IN)                      :: RG_TYPE(:)  ! regrid type
    LOGICAL, INTENT(IN)                      :: lint        ! input time ?
    TYPE (t_ncvar), DIMENSION(:), POINTER    :: invar ! list of input  variables
    TYPE (t_ncvar), DIMENSION(:), POINTER    ::   var ! list of output variables
    TYPE(t_geohybgrid), INTENT(INOUT), OPTIONAL :: grid
    LOGICAL, INTENT(IN)              , OPTIONAL :: llrgz
    LOGICAL, INTENT(IN)              , OPTIONAL :: lfirsto  ! first output step
! ------------------------------------------------------------------------------
```

Arguments to the subroutine are

- a status flag `status` indicating success or failure of the subroutine.
- the ID `SCRIP_ID` of the SCRIP data set to be used.
- the input grid `igrid`.

- the output grid `ogrid`.
- the interpolation type `RG_TYPE`, i.e., one of `INT`, `EXT`, `IDX` or `IXF`.
- the logical `lint`, indicating if the input or the output grid time should be use: `lint=.TRUE.` means input time is used.
- the list of input variables `invar`.
- the list of output variables `var`.
- optionally a grid variable (`grid`), on which, if present, the final unsorted, completed grid is copied.
- an optional logical (`llrgz`), indicating if the variable should also be vertically interpolated. As SCRIP does only horizontal interpolations this information is required for the file export of the interpolated data. In case the variable will be also vertically interpolated the output must not be performed in `SCRIP_CONTROL`, but in `REGRID_CONTROL`.
- the logical `lfirsto` is also required for the output of the regridded data. It indicates, whether this is the first output step. This information is necessary, as the attributes can be written only once to the output file.

`SCRIP_CONTROL` starts with the initialisation of some local and the output variables.

The workflow of `SCRIP_CONTROL` is as follows:

- Preparation of grids and variables for the regridding:
  - Grid operations:
    - Creation of local copies `gi` and `gg` of input and output grid (`igrid` and `ogrid`), respectively.
    - Check of input grid `gi` by calling `CHECK_GEOHYBGRID`.
    - Switching of the vertical axis to an invariant axis in the output grid `gg` by calling `SWITCH_GEOHYBGRID` in order to trigger purly horizontal interpolation.
    - Check of output grid `gg` by calling `CHECK_GEOHYBGRID`.
    - Balancing of the time axes of `gi` and `gg` by calling `BALANCE_GEOHYBGRID_TIME`, i.e., both grids are adjusted to the same time, i.e., the input grid time if `lint=.TRUE.`, otherwise the output grid time is used.
    - Copying of the input grid `gi` to the sorted grid `gis`. The grids are automatically sorted by `SWITCH_GEOHYBGRID` call.
    - Copying of the ouput grid `gg` to the sorted grid `ggs`. The grids are automatically sorted by `SWITCH_GEOHYBGRID` call.
    - Completion of the "sorted" input grid `gis` by calling `COMPLETE_GEOHYBGRID`. In this subroutine missing components for interfaces or mid-point variables are calculated from the other component, i.e., interfaces from mid-point or mid-point from interfaces. If interfaces and mid-points are not available, these components stay undefined.
    - Completion of the "sorted" output grid `ggs` by calling `COMPLETE_GEOHYBGRID`.
  - Variable operations:
    In a loop over the variables (indexed `i`) to be regridded the following preparations are made:

  - Call of subroutine `IDX2FRAC_NCVAR` for preparation of input variable, if `IXF` or `IDX` regridding is requested (see section 3.1.2.12). In this case `IDX2FRAC_NCVAR` outputs the new local variable `qvari`. Otherwise, copying of `invar(i)` to the new local variable `qvari`.
  - Check of conformity of variable `qvari` with the sorted input grid `gis` definition by calling `CHECK_NCVAR_ON_GEOHYBGRID`.
  - Balancing of the output variable `xovar(i)` with input variable `qvari` and sorted output grid `ggs` by calling `BALANCE_GEOHYBGRID_NCVAR` (see Sect. 3.3.10).
  - Balancing of the sorted output variable `svaro(i)` with input variable `qvari` and sorted output grid `ggs` by calling `BALANCE_GEOHYBGRID_NCVAR` (see Sect. 3.3.10).
  - Packing of sorted input variable `qvari` by calling `PACK_GEOHYBGRID_NCVAR`. The packed variable is called `pvari`.
  - Balancing of packed output variable `pvaro(i)` with packed input variable `pvari` and sorted output grid `ggs` by calling `BALANCE_GEOHYBGRID_NCVAR` (see Sect. 3.3.10).
  - Deallocation / Initialisation of the data component of `pvaro(i)` by calling `INIT_NARRAY(pvaro(i)%dat)`.
  - Deallocation / Initialisation of local input variables `qvari` and `svari` by calling `INIT_NCVAR`.

  After this is done for every variable to be interpolated, the data is ready for interpolation.

- SCRIP interpolation:
  The interpolation via SCRIP is simply performed by calling the subroutine `APPLY_SCRIP_WEIGHTS` (Sect. 3.8.15).

- Reformatting of output variables and grids:
  After the interpolation the data needs to be transformed back to its requested order.

  – In a loop over all variables (loop index `i`) the variables are transformed back into their requested form:
    - Check of the sorted variable `svaro(i)` on the sorted output grid `ggs` by calling `CHECK_NCVAR_ON_GEOHYBGRID` to get the definition of output axes (`axes`) and dimensions (`dims`).
    - Unpacking of packed output variable `pvaro(i)` using `axes` and `dims` by calling `PACK_GEOHYBGRID_NCVAR`. Output of this subroutine is the sorted output variable `svaro(i)`.
    - For 'IDX' interpolated variables, transformation of the data by calling the subroutine `MAXFRAC2IDX_NCVAR`, otherwise copying of `svaro(i)` to the final output variable `xovar(i)`.
    - Finally in the loop over the variables, initialisation of the local variables `pvari(i)`, `pvaro(i)`, `svaro(i)`, `qvaro`.

  – Memory deallocation:
    The variables `pvari`, `pvaro` and `svaro` are initialised and deallocated and the corresponding pointers are nullified.

  – If a filename is defined in the grid structure `gg` and `lrgz == .FALSE.`, direct output of the interpolated fields is requested. First, `gg` is written to the file by the subroutine `EXPORT_GEOHYBGRID`. Second, if called for the first time (`lfirsto = .TRUE.`), the attributes are added to the file. Finally, the variables are written using `EXPORT_NCVAR`.

  - Preparation of the returned data:
    The output variable `var` is allocated to the number of interpolated variables and the content of the local output variable `xovar` is copied to `var`. If the output grid is required, `ggs` is copied to the output subroutine argument `grid`.

- At the end, internal memory is released.

3.8.17 INTERPOL_GEOHYBGRID_PS

This subroutine basically uses the same algorithm as `SCRIP_CONTROL` to horizontally interpolate the surface pressure defined on one grid to a surface pressure field on another grid.

As this subroutine calls `APPLY_SCRIP_WEIGHTS`, it is only applicable within SCRIP.

3.8.18 BALANCE_CURVILINEAR_PS

This subroutine adjusts the surface pressure definition of two grids to each other.

First, some consistency checks have to be performed:

- In case none of the two grids contains a surface pressure definition (`ps`), the hybrid-b coefficients must also be undefined.
- In case none of the two grids contains a reference pressure definition (`p0`), the hybrid-a coefficients must also be undefined.
- The subroutine returns, if the grid is 2D, i.e., neither the surface or the reference pressure, nor any hybrid coefficients are defined.

If the consistency checks are passed, the grids can be balanced. If `p0` is defined on one of the grids and not on the other, the defined reference pressure is copied to the other grid. For the surface pressure itself, four different cases can occur:

A) `ps` is defined for both grids. Thus only the time axis of both grids need to be adjusted, but this is done in the subroutine `BALANCE_GEOHYBGRID_TIME`.

B) `ps` is undefined in both grids. In this case nothing needs to be done.

C) the surface pressure of the outgoing grid `go%ps` is defined, but on the wrong grid.
In this case an error message is produced and the interpolation is interrupted.

D) If the incoming or the outgoing surface pressure are defined, the missing surface pressure is constructed by calling the subroutine `INTERPOL_GEOHYBGRID_PS` (Sect. 3.8.17).

As `INTERPOL_GEOHYBGRID_PS` calls `APPLY_SCRIP_WEIGHTS`, this subroutine is only applicable for SCRIP interpolation.

3.8.19 CONSTRUCT_INPUT_SURF_PRESSURE

The subroutine `CONSTRUCT_INPUT_SURF_PRESSURE` helps to construct a surface pressure for the input grid. Currently this subroutine is called from the module `MESSY_MAIN_IMPORT_GRID` to construct a surface pressure for the grid which is, after horizontal interpolation via SCRIP, vertically interpolated by NREGRID. Input parameter to this subroutine are the input grid `gi` of type `t_geohybgrid`, which was already input to `SCRIP_CONTROL` and the SCRIP data set ID `PSDID`. Additionally, the intermediate grid (i.e., after horizontal interpolation, before vertical interpolation) `gips`, which requires the newly constructed surface pressure variable to be `INTENT(INOUT)`, while a `status` flag informs about success or failure of the subroutine.

If the input grid component `ps` is not defined for the input grid, a surface variable `ps` is constructed from the dimensions of the longitude and latitude axes of the `gips` grid. As no information about the actual pressure is available, `ps` is set to `101325 Pa` everywhere.

If `gi%ps` is defined, the surface pressure is interpolated from the input to the intermediate grid by calling `INTERPOL_GEOHYBGRID_PS`.

As this subroutine calls a subroutine, which is only applicable for SCRIP, this subroutine is also only applicable for SCRIP.

3.9 MESSY_MAIN_GRID_MPI

As both, the IMPORT_GRID stand-alone tool and GRID implemented in a 3D model, can be applied in (distributed memory) parallel decomposition, it is important, that in case of an error the model is aborted correctly, i.e., by calling `MPI_ABORT`. There are two ways to implement such a model abort. Either handing back status flags to the highest model layer and aborting the model from there, or, calling the model abort directly.

The second way was chosen for NREGRID and thus now for GRID. `MESSY_MAIN_GRID_MPI` contains the abortion routines `GRID_ABORT` and `P_ABORT`. `GRID_ABORT` writes an error file (named "END"), which is specifially required, if GRID is run within the MESSy infrastructure, as the universal runscript `xmessy_mmd` interrupts the job chain, if a file named "END" exists. At the end the subroutine `P_ABORT` is called.

`P_ABORT` calls `MPI_ABORT` for the communicator `MPI_COMM_WORLD`, thus terminating the simulation on all PEs associated with the job.

4 The BMIL GRID files

Most of the grid definition and transformation is done in the SMCL layer of the submodel. Nevertheless, GRID can be run in parallel environments and higher order models. For the parallel environment it might be necessary, depending on the implementation, that a grid definition is only performed on one processor. In this case the grid needs to be broadcasted to the other processors. This functionality is provided by the subroutine `P_BCAST_GRID` in the module `MESSY_MAIN_GRID_BI`: This subroutine requires itself subroutines for broadcasting the components of a grid. They are located in the module `MESSY_MAIN_GRID_NETCDF_BI`.

Additionally in `MESSY_MAIN_GRID_BI`, for a multi-dimensional model a so-called basemodel grid is defined, which is the reference grid.

4.1 MESSY_MAIN_GRID_BI

The module `MESSY_MAIN_GRID_BI` is the interface file linking the core of the submodel GRID to a multi-dimensional model. Apart from the exchange of grid definitions between parallel running talks, a standard target (default) basemodel grid is defined. A pointer to the basemodel grid `bgrid_ptr` and the ID of the grid definition in the list of grids (`BASEGRID_ID`) are defined in `MAIN_GRID_INIT_MEMORY` and are made available throughout the simulation / model. While the pointer to the basemodel grid (`bgrid_ptr`) is defined in `MESSY_MAIN_GRID_BI`, for avoiding circular dependencies in the COSMO/MESSy model, the `BASEGRID_ID` is defined in the module `MESSY_MAIN_DATA_BI`.

4.1.1 P_BCAST_GRID

In case of a parallel processing of the model, it might be intended that only one task (or PE) is gathering the information of a grid at first. If thereafter the grid is to be known on all PEs, the grid, i.e., a variable of type `t_geohybgrid` needs to be broadcasted. This is what `P_BCAST_GRID` is doing. To go into a little more detail, input to this subroutine are the variable of type `t_geohybgrid` to be broadcasted (`grid`) and the ID of the sending PE (`proc`). First, the `grid` is initialised on all PEs except the one with ID `proc` by calling the subroutine `INIT_GEOHYBGRID`. Secondly, all components of `grid` are broadcasted by calling the subroutines `P_BCAST` or `P_BCAST_NCVAR`.

4.1.2 MAIN_GRID_INIT_MEMORY

This subroutine is called during the initial phase of a model simulation. It primarily contains the definition of the basemodel grid in a variable of type `t_geohybgrid`. This subroutine needs to be called prior to the memory allocation in other submodels. Specificially, this subroutine needs to be called prior to `MAIN_IMPORT_INIT_MEMORY`, as the base grid definition is required there.

Depending on the basemodel, `MAIN_GRID_INIT_MEMORY` needs to be called once or twice. The mandatory call is the one with `flag=2`. In this call, the basemodel grid is defined and the global identifiers, i.e., the ID of the basemodel grid (`BASEGRID_ID`) and the pointer to the basemodel grid (`bgrid_ptr`) are set accordingly. At the moment, `GRID` is used within two 3D models: EMAC and COSMO/MESSy. For the EMAC model the call with `flag==2` is sufficient, while for the COSMO model an extra call is required, because the vertical grid is not fully set up at this point in time during the model initialisation. Therefore the information of the vertical grid needs to be acquired earlier during the initialisation (i.e., here) in addition to the usual place where it is read in COSMO. Depending on the

mode of operation of COSMO/MESSy (stand-alone or on-line nested into EMAC or COSMO/MESSy) the information needs to be read from different files: In the on-line nested mode, the information comes from the INT2COSMO namelist `INPUT.nml`, while in the stand-alone mode the information is part of the initial or boundary files. Therefore, the subroutine `GRID_READ_VERTAXIS`, which is contained in `MAIN_GRID_INIT_MEMORY` calls different reading procedures depending on the mode of operation.

4.1.3 MAIN_GRID_FREE_MEMORY

The subroutine `MAIN_GRID_FREE_MEMORY` is called in the finalising phase of a model simulation. It calls the subroutines `CLEAN_SCRIPDATA_LIST` and `CLEAN_SCRIPGRID_LIST` (both located in the module `MESSY_MAIN_GRID_TRAFO_SCRP`) to free the memory that has been allocated during the intialisation phase for the grid and data definition and destroy the concatenated lists.

4.2 MESSY_MAIN_GRID_NETCDF_BI

If a grid was only read on one PE, it must be broadcasted to the other PEs. The module `MESSY_MAIN_GRID_NETCDF_BI` contains the broadcasting subroutines for the structure elements of `t_geohybgrid`, which are defined in `MESSY_MAIN_GRID_NETCDF`. They are based on and handle netCDF specific structures.

4.2.1 P_BCAST_NCVAR

This subroutine broadcasts a variable of type `t_ncvar`.

4.2.2 P_BCAST_NCATT

This subroutine broadcasts a variable of type `t_ncatt`.

4.2.3 P_BCAST_NCDIM

This subroutine broadcasts a variable of type `t_ncdim`.

4.2.4 P_BCAST_NARRAY

This subroutine broadcasts a variable of type `t_narray`.

References

Jöckel, P.: Technical note: Recursive rediscretisation of geo-scientific data in the Modular Earth Submodel System (MESSy), Atmos. Chem. Phys., 6, 3557–3562, 2006.

Jones, P.: First- and Second-Order Conservative Remapping Schemes for Grids in Spherical Coordinates, Mon. Wea. Rev., 127, 22042210, 1999.