IMPORT 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 generic 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

Chapter 1

Introduction

This is the User Manual for the generic MESSy submodel IMPORT. IMPORT currently consists of two submodels: IMPORT_GRID and IMPORT_TS. Both are implemented as part of the MESSy infrastructure and thus available in all MESSy 3D models (esp. in EMAC and the COSMO model). This application is called the interface mode in the following. In addition, both are available as stand-alone tools. In the interface mode the IMPORT namelist file `import.nml` controls IMPORT_GRID (Sect. 2) and IMPORT_TS (Sect. 3). Each submodel uses its own namelist, which are described in the respective sections (Sects. 2.5 or 3.1). Only these namelists need to be modified if a new simulation is set up based on already existing import data.

To implement a new import field in IMPORT_GRID requires, in addition to the modification of the `import.nml` namelist file, the provision of a so-called `&regrid` namelist. Their definition is explained in Sect. 2.5. The second and the third part of the User Manual are dedicated to the submodels IMPORT_GRID (Sect. 2) and IMPORT_TS (Sect. 3), respectively.

The code of the stand-alone tools is available under the GPL license (Sect. 1.0.1). For the SCRIP software (Jones, 1999) implemented into IMPORT_GRID the license terms of SCRIP apply to this part of the code (Sect. 1.0.2).

1.0.1 IMPORT and NREGRID license

1.0.2 SCRIP license

For **SCRIP** the regulations of SCRIP apply (cited from SCRIPusers.pdf distributed with SCRIP v1.4):

Chapter 2

IMPORT_GRID

2.1 Prerequisites

IMPORT is written in Fortran95, thus a Fortran95 compiler is required for installation of the software. (A Fortran90 compiler which is capable to handle initialisation statements in declaration lines should do as well.) Furthermore, (g)make is used for building the software. Since the input and output format is netCDF (http://www.unidata.ucar.edu/ packages/netcdf/), Fortran90 bindings (i.e., at least netCDF - Version 3.6.0) of the netCDF library are required. IMPORT comprises the f2kcli command line interface (http://www.winteracter.com/f2kcli/index.htm) for the stand-alone tools.

2.2 Installation of the stand-alone tool IMPORT_GRID

The installation is straight forward.

1. unzip the zip-file:
   `unzip import_grid.zip`
2. change into the subdirectory `./import_grid`:
   `cd import_grid`
3. configure IMPORT_GRID according to your system:
   `./configure` VAR_1=VAL_1 VAR_2=VAL_2 `...` :

   where possible *VAR(IABLE)s* are

`F90`	Fortran90/95 compiler	(optional)
`F90FLAGS`	Fortran90/95 compiler options (e.g., options for invoking the cpp preprocessor	(optional)
`NC_INC`	(absolute) path where 'netcdf.mod' is located	(required)
`NC_LIB`	(absolute) path where 'libnetcdf.a' is located	(required)

   Platform and compiler specific notes can be found in the README file of the distribution.
4. build the executables and modules:
   `gmake`
5. move the executable and the modules to the `./bin` and `./include` subdirectories, respectively:
   `gmake install`

6. the directory can be cleaned by
   `gmake clean`

The executable `import_grid.exe` should now be available in the `./bin` subdirectory and the modules are in the `./include` subdirectory. The status after unpacking the zip-file can be reset with `gmake distclean`.

2.3 Usage

IMPORT_GRID can be applied in two different modes:

- The stand-alone mode for the rediscretisation of data files in netCDF.
- The interface mode (coupled to a model) for automatic rediscretisation during data import from netCDF files.

In both modes, IMPORT_GRID is basically applied in 4 steps:

1. Analysis of the netCDF file containing the data which should be regridded (*infile*). This can for instance be done with:
   `ncdump -h` *infile*
   Required are the names of the netCDF variables spanning the grid (such as latitude, longitude, surface pressure, reference pressure, hybrid-coefficients, time), and the names of the variables which should be regridded. Note that this step is left to the user, in order to be independent of specific netCDF conventions. Note further, that netCDF and IMPORT_GRID are case-sensitive.

2. For the analysis of the output grid structure three ways exist, depending on the application mode:

   - The output grid structure is available in a netCDF file (called *grdfile*):
     The information can be extracted from the $grdfile$ in analogy to step 1. above.
   - IMPORT_GRID is used in interface mode:
     The output grid information is provided by the grid-specification interface routine (see Sect. 2.5.1).
   - The output grid structure is not available:
     The output grid information can be written with an appropriate text editor in the CDL-syntax (see netCDF manual). From this, a netCDF file can be generated by
     `ncgen -b -o` *grdfile cdl-file*
     and used as *grdfile*, as above.

3. Summary of all required information in a namelist:
   The IMPORT_GRID namelist structure is described in detail in Sect. 2.5.

4. Start IMPORT_GRID:

   - start IMPORT_GRID in stand-alone mode:
     `import_grid.exe` *namelist-file*
   - start IMPORT_GRID in interface mode:
     start the executable with IMPORT_GRID linked

Detailed information about the usage of IMPORT_GRID from Fortran90 / Fortran95 code is provided in Sect. 2.6.

2.4 Regridding Types

IMPORT_GRID contains two different software packages for grid transformation, NREGRID (Jöckel, 2006) and SCRIP (Jones, 1999). NREGRID is designed to rediscretise arbitrary distributions from one grid to another, without adding information (e.g., if regridding from a coarser to a finer grid), or reducing more information than is lost anyway (e.g., if regridding from a finer to a coarser grid). Thus, no inter- / extrapolation methods are applied! The data fields are purely redistributed, assuming constant values within the grid box, represented by the grid box mid-point. For this, intensive and extensive variables are distinguished. Accordingly, the following regridding types (RG_TYPE) are supported by NREGRID at present for the regridding of variables (scalar fields!) between geo-hybrid grids:

- INT is suitable for intensive quantities, such as tracer mass mixing ratios, or temperature fields. This option conserves the global area/volume weighted average of a scalar field during the regridding procedure.
- EXT is suitable for extensive quantities, such as tracer masses, or tracer emission maps (2D) in units "per box". This option conserves the global unweighted sum over all grid boxes of a scalar field.
- IDX is suitable for regridding index distributions, i.e., scalar fields with discrete values, where averaging is not defined. The index-regridding returns for a given destination grid-box the index, which has in all overlapping source grid-boxes the largest relative contribution.
- IXF is similar to IDX, however it returns a variable extended by one dimension, whereby the additional dimension is along the index range. A given slice along the new index-dimension contains the fraction of that index in the respective grid-box.

Apart from the EXT type, these types are also available for transformations by SCRIP.

2.5 Namelist control

The regridding procedure of IMPORT_GRID is controlled by a namelist. In interface mode additional control is provided by optional parameters specified at the subroutine calls of `REGRID_CONTROL` or `SCRIP_CONTROL`, respectively. The syntax of an IMPORT_GRID namelist is:

```
!========================
&REGRID
variable = value,
...
/
!========================
```

`variable` is the name of the namelist variable and `value` its assigned value. The dots indicate a list of further namelist entries. Several namelists can be concatenated into one namelist file. Those are sequentially processed by IMPORT_GRID. Table 2.1 gives an overview of the possible namelist entries with their meanings.

INPUT	OUTPUT	TYPE	Description
infile	*outfile*	CHAR	input/ output netCDF file
grdfile		CHAR	netCDF file with output grid
`i_lat(m/i)`	`g_lat(m/i)`	CHAR	name of latitude axis
`i_latr`	`g_latr`	REAL, DIM(2)	range of latitude axis
`i_lon(m/i)`	`g_lon(m/i)`	CHAR	name of longitude axis
`i_lonr`	`g_lonr`	REAL, DIM(2)	range of longitude axis
`i_lonc`	`g_lonc`	LOGICAL	longitude axis is modulo axis
`i_hya(m/i)`	`g_hya(m/i)`	CHAR	name of hybrid-A-coefficient (h_a)
`i_hyar`	`g_hyar`	REAL, DIM(2)	range of hybrid-A-coefficient
`i_hyb(m/i)`	`g_hyb(m/i)`	CHAR	name of hybrid-B-coefficient (h_b)
`i_hybr`	`g_hybr`	REAL, DIM(2)	range of hybrid-B-coefficient
`i_time(m/i)`	`g_time(m/i)`	CHAR	name of time axis
`i_ps`	`g_ps`	CHAR	name / value of surface pressure
`i_p0`	`g_p0`	CHAR	name / value of reference pressure
`i_t`	`o_t`	INTEGER, DIM(X)	input (X=3)/output(X=4) time step control
`g_t`		INTEGER, DIM(3)	grid time step control
`i_clat(m/i)`	`g_clat(m/i)`	CHAR	name of geographical latitude coordinate of curvi-linear grids
`i_clon(m/i)`	`g_clon(m/i)`	CHAR	name of geographical longitude coordinate of curvi-linear grids
`i_clonc`	`g_clonc`	LOGICAL	longitude axis of curvi-linear grid is modulo axis
`i_rlat(m/i)`	`g_rlat(m/i)`	CHAR	name of rotated latitude axis for rotated grids
`i_rlon(m/i)`	`g_rlon(m/i)`	CHAR	name of rotated longitude axis for rotated grids
`i_rlonc`	`g_rlonc`	LOGICAL	longitude axis of rotated grid is modulo axis
`i_pollon`	`g_pollon`	REAL	longitude of rotated North Pole of rotated grid
`i_pollat`	`g_pollat`	REAL	latitude of rotated North Pole of rotated grid
`i_polgam`	`g_polgam`	REAL	angle between the north poles of the rotated and the geographical grids
`var`		CHAR	variable list
`pressure`		LOGICAL	vertical regridding in pressure coordinates
`input_time`		LOGICAL	output file gets input time axis

Table 2.1: List of namelist variables of &RGTEVENT namelists. `(...m/...i)` denote the box mid-point `(...m)` and interface `(...i)` coordinates, respectively.

2.5.1 Grid specification

In order to allow netCDF files to be as generic as possible, and not to be restricted to specific netCDF conventions, the input grid (read from the netCDF file *infile*) has to be specified by the user via the namelist. With the namelist variables `i_lat(i/m)`, `i_lon(i/m)`, `i_hya(i/m)`, `i_hyb(i/m)`, `i_ps`, and `i_p0` a geographically-rectangular, 3D input grid (spatial) is fully described. More complex horizontal grids can be described by `i_clat(i/m)` and `i_clon(i/m)`. These strings provide the names of the variables defining the geographical longitude and latiude for a curvi-linear grid. In the specific case

of rotated rectangular grids, additionally, `i_rlat(i/m)` and `i_rlon(i/m)` can be defined. These variables name the longitude and latitude axis in the rotated system. For transformation between rotated and geographical coordinates, the definition of the rotated pole is required. This is defined by `i_pollon`, `i_pollat` and `i_polgam`. For the special case, that the longitude axis is a modulo axis specific assumptions can be made in the algorithm. Therefore this information needs to be provided by the logical variables `i_lonc`, `i_clonc` and `i_rlonc`, respectively.

The output grid `(g ...)` is specified in analogy, and read from the *grdfile* in the stand-alone mode, while it is provided by the basemodel in interface mode. The output is written to the netCDF file *outfile*. For the regridding procedure, the interfaces `(i...)` of the grid boxes are required (except for `i/g_timei`). If the respective data are not available in the *infile* / *grdfile*, or the entries are not present in the namelist, the interface values are (for geographically-rectangular grids) internally calculated from the corresponding mid-box values, assuming that the interfaces are half-way between the mid-points. The outer interfaces are calculated using the same distance between outermost mid-point and corresponding inner interface. If this calculation of the outer interfaces is not applicable, the user can specify them with the namelist variables `i_latr`, `i_lonr`, `i_hyar`, `i_hybr` for the input grid, and with `g_latr`, `g_lonr`, `g_hyar`, `g_hybr` for the output grid, respectively. For example,

```
...
i_latm = latitude,
i_latr = -90.0, 90.0,
...
```

in the namelist ensures that the outer input grid latitude interfaces (calculated from the mid-box latitudes with name latitude) are at -90.0° and 90.0°. Note that in case of `i/g_hyar` the order of parameters is relevant, since the hybrid-A-coefficients (see Eq. 2.1 below) are not monotone. Therefore, in
`i_hyar = a1, a2,`
`a1` refers to the highest grid level (corresponding to the smallest hybrid-B-coefficient (!)), and `a2` to the lowest grid level (corresponding to the largest hybrid-B-coefficient (!)), respectively.

If the interface variables are specified in the namelist, but not the corresponding mid-points, the latter are internally calculated, assuming that the mid-points are half-way between the interfaces. The mid-points, however, are not used by the regridding procedure. Dimensions defined for the input grid (*infile*), but omitted for the output grid (*grdfile*) are treated as invariant (see GRID-User-Manual).

2.5.2 Syntax of the namelist variable `var`

With the namelist variable `var` the user specifies the scalar field contained in *infile* (which have to be on the specified input grid) that should be regridded. For output to the *outfile* (in stand-alone mode) or to the interface, the variables can be renamed and scaled. Moreover, the regridding type (see Sect. 2.4) can be assigned. The syntax is
`var = '[new_name=]name[:RG_TYPE][,scale]; ...',`
whereby the dots indicate a list of further variables. `name` is the variable name in *infile*, `new_name` is the variable name in *outfile*, `RG_TYPE` is the regridding type (see Sect. 2.4), and `scale` is the scaling factor. The order of `:RG TYPE` and `,scale` is arbitrary. If `new_name` is omitted, the variable is not renamed. If `scale` is omitted, the variable is not scaled. If `RG_TYPE` is omitted, IMPORT_GRID checks the *infile* for the variable attribute `RG_TYPE`. If this attribute is not set, or the value is not recognised, IMPORT_GRID takes `INT` as default (see Sect. 2.4). If the namelist variable `var` is not specified at

all, IMPORT_GRID scans the *infile* for all variables on the specified input grid. Renaming and scaling are not performed. The regridding type is set to `INT`, unless the variable attribute `RG_TYPE` in *infile* is specified.

2.5.3 Time control

With the time control namelist variables `i_t`, `g_t` and `o_t` the user specifies the time steps of netCDF variables for regridding. The syntax is

```
i_t = itmin,itstep,itmax,itret, ! default: 1, 1, 0, 0
g_t = gtmin,gtstep,gtreset,     ! default: 1, 1, 0
o_t = otstart,otstep,otdummy    ! default: 1, 1, 0
```

where *itmin, itstep, itmax, itret, gtmin, gtstep, gtreset, otstart*, and *otstep* are integers. The default settings are listed above. The third entry of `o_t` (*otdummy*) is currently not used. IMPORT_GRID resets automatically *itstep, gtstep*, and/or *otstep* to 1, if 0 is specified in the namelist; *itmax* and *itret* are set to *itmin*, if not specified in the namelist. The overall consistency of all time step control parameters is checked by IMPORT_GRID, and documented by the output of error / warning messages, if required. The input variables are regridded between *itmin* and *itmax* with a step size of *itstep*. The regridded data of time step *itret* are returned to the `READ_CONTROL` subroutine call (see Sect. 2.6.3). Thus, *itret* has no meaning in the stand-alone mode of IMPORT_GRID. For the destination grid of the variables at the input time steps (*itmin*, *itmin+itstep*, *itmin+2itstep , ... , itmax*) the grid from *grdfile* is used at time step *gtmin, gtmin+gtstep, gtmin+2gtstep, ...* , respectively. If *gtreset* is reached, the *grdfile* time step is reset to *gtstart* again, etc. (This allows for instance the regridding of 60 time steps of monthly averaged data (in *infile*), to a grid (in *grdfile*) which is only known climatologically, i.e., containing 12 monthly averages.) And finally, the regridded data is written to the output file with time steps *otstart, otstart+otstep, otstart+2otstep, ...* . This is needed, e.g., if *itstep* is not 1, but the *outfile* should contain a continuous time series. With the namelist variable `input_time` the time axis specification in the output file (*outfile*) is set. Per default (`input_time = .TRUE.`) the *outfile* time axis is the same as in *infile*. Otherwise, (`input_time = .FALSE.`) the *grdfile* time axis (or interface time axis in interface mode) is taken for *outfile*. Additionally, in interface mode the output time stepping (`o_t`) is set to the *infile* time stepping (in case of `input_time= .TRUE.`), or to the interface time stepping (in case of `input_time = F`), respectively.

2.5.4 Vertical axis specifications

As described earlier, SCRIP is a purely horizontal grid transformation algorithm. Therefore the vertical regridding is always performed using NREGRID. NREGRID is capable to handle all cases of vertical pressure axes of the form:

$$p(x, y, z, t) = h_a(z) \cdot p_0 + h_b(z) \cdot p_s(x, y, t), \tag{2.1}$$

such as

- hybrid pressure axes ($h_a \neq 0$, $h_b \neq 0$);
- constant pressure axes ($h_b = 0$); `i/g_hybi/m` omitted in namelist
- sigma levels ($h_a = 0$); `i/g_hyai/m` omitted in namelist

2.5.4.1 Surface and reference pressure

If the surface pressure and/or reference pressure is / are not contained in *infile* and / or *grdfile*, respectively, or they should not be used, it is possible to specify a constant value, for example:

```
i_p0 = '101325.0 Pa'
```

The syntax is the same for `g_p0`, `i_ps`, and `g_ps`. Note that in these cases the unit must be chosen such that surface pressure (p_S), reference pressure (p_0) and the hybrid-coefficients (h_a, h_b) are consistent because of the given relationship for the vertical pressure (p), dependent on longitude (x), latitude (y) and time (t) at a hybrid level (index i):

$$p(i, x, y, t) = h_a(i) \cdot p_0 + h_b(i) \cdot p_s(x, y, t). \tag{2.2}$$

The unit (Pa in the specification above) is only converted to the netCDF variable attribute units in the output file, but not used internally for automatic unit conversions!

2.5.4.2 Vertical regridding coordinates

Calculation of the vertical overlap of grid-boxes between *infile* and *grdfile* is internally performed in sigma-coordinates per default

$$\sigma(i) = p(i, x, y, t)/p_s(x, y, t), \tag{2.3}$$

in order to avoid conservation problems in case the source and destination surface pressure fields are different. However, a vertical regridding in pressure coordinates can be enforced, if the variable `pressure = T` is specified in the namelist. Note, however, that in such cases input and output pressure levels must have the same units!

2.5.5 Namelist examples

Examples of IMPORT_GRID namelists can be found in the `./import_grid/nml` subdirectory of this distribution.

2.6 Interface Mode

The usage of IMPORT_GRID in interface mode, i.e., linked to another program, requires two steps:

- Specification of the destination grid in the Fortran95 code, and
- calling the regridding procedure from the Fortran95 code. This can be achieved by using the interface routines described below.

2.6.1 Destination grid specification

For IMPORT_GRID in interface mode (i.e., as part of a model) a specification of the destination grid in the Fortran95 code is required, which provides the information as alternative to the specification via the *grdfile* entry in the namelist (see Sect. 2.5). The definition of the basemodel grid is provided by the MESSy submodel GRID (see the GRID-User-Manual available in the same supplement as this manual). The IMPORT_GRID interface locates this grid information by the `BASEGRID_ID` provided by GRID. Alternatively, a destination grid defined by any other MESSy submodel can be chosen by namelist entry (see Sect. 2.6.2)

2.6.2 IMPORT_GRID namelist control in interface mode

In Sect. 2.5 the namelist required for one specific mapping process is explained. However, IMPORT_GRID in interface mode can deal with an arbitrary number of different grid transformations at different time steps during one simulation. Each of those is called a *regrid event* in the following. The individual *regrid events* are defined in a namelist called `&RGTEVENTS`. The specific entries are labeled with `RG_TRIG` (for regridding trigger) followed by a unique number.

```
&RGTEVENTS
! ### SYNTAX:
! ###         NML=   '' (DEFAULT)   : this namelist-file (import.nml)
!                    <namelist file>: other namelist file
!
! ###         FILE=  '' (DEFAULT)   : - ONLY first netCDF file in NML
!                    <netCDF-file>  : - this file in NML
!
! ###         VAR=   '' (DEFAULT)   : - all variables from FILE
!                                     - all variables in first namelist in NML
!                    <tracer name>  : - this variable from namelist in NML
!                                       (FILE specifier ignored !!!)
! ###         Z=     <z1,z2,...>    : - list of emission heights [m]
!                                       (above GND) for multi level emissions
!                                       (Nx2D)
!
!           {--------EVENT---------} {--------------------STEPPER ----------------------}
!                                    {------counter-------}{----------action string-----}
!
! GHG: N2O, CH4, CO2  JAN-1950:   1   DEC-2011: 744
RG_TRIG(1)  = 1,'months', 'first',0, 'GHG',  733,1,744,733, 'NML=./import/GHG.nml;',
! --------------------
! CCMI biomass burning
! --------------------
!
!  JAN-1950:   1   DEC-2010: 732
!
RG_TRIG(2)=1,'months','first',0,'NMHC',721,1,732,709,'NML=./import/NMHC_BB.nml; Z=50,300; GRID=TEST2',
!
! VOLCANIC SO2
!
RG_TRIG(10)=1,'months','first',0,'VOL_SO2',1,1,12,1, 'NML=./import/volc_SO2.nml; VAR=SO2;IPOL=SCRP',
/
```

Each `RG_TRIG` entry consists of an *event* and a *stepper*. The *event* defines a periodically occuring event[1], e.g., for the regridding of an emission field every month. The *stepper* consists of a *counter* and an *action string*. The *counter* is defined as:

counter = name, min, step, max, start

It defines the cyclic stepping through the time steps of the netCDF input file. *name* is a string defining a name by which the *counter* can be identified. *start* is the initial value of the *counter* at the very first model time step. During the simulation the *counter* is increased by *step* until *max* is reached. Afterwards the *counter* is reset to *min*. In the example namelist, the *counter* with the name NMHC

[1]For the exact definition of an event see the User Manual of the generic MESSy submodel TIMER, which is part of the electronic supplement of Jöckel et al., 2010; http://www.geosci-model-dev.net/3/717/2010/

starts with 709, is incremented by 1 at the beginning of each month until 732 is reached, then the *counter* is reset to 721.

Finally, the *action string* controls remapping specific features. In the IMPORT_GRID *stepper action string* the following keywords, separated by semicolons, are recognized:

keyword	meaning
`NML=`	followed by the name (including the path) of the file containing the `&REGRID` namelist for grid import. If this keyword is omitted (or empty), the IMPORT namelist file itself (`import.nml`) is used.
`FILE=`	followed by the name (including the path) of the input file. IMPORT_GRID loops over all `&REGRID` namelists in the specific namelist file (`NML=...`), until the first namelist with matching netCDF filename is found. If this keyword is omitted (or empty), the first namelist in the specified namelist file (`NML=...`) is used.
`VAR=`	followed by the name of the variable that is imported by IMPORT_GRID. IMPORT_GRID loops over all IMPORT_GRID namelists in a specific namelist file (the result of `NML=`), until the first namelist with matching variable name is found (in this case, the `FILE` specifier is ignored). If this keyword is omitted, IMPORT_GRID imports all variables from `FILE`, if specified, or all variables from the first `&REGRID` namelist in `NML`.
`Z=`	followed by a comma separated list of geometric heights (in meter above ground). This is only applicable to multilevel (Nx2D) data. The number of heights must match the number of levels (N) in the input file.
`IPOL=`	identifies the interpolation method. It can be one of `SCRP` for SCRIP or `NRGT` for NRE-GRID remapping or `NONE` for raw data import.
`GRID=`	is a string identifier for the output grid. The grid of the given name needs to be defined somewhere in the 3D model, otherwise IMPORT_GRID terminates the simulation.

2.6.3 The IMPORT_GRID BMIL, Tools and SMCL

IMPORT_GRID consists of four Fortran90 modules / files[2]:

1. The BMIL[3] module `MESSY_MAIN_IMPORT_GRID_BI`
2. The BMIL module `MESSY_MAIN_IMPORT_GRID_TOOLS_BI`
3. The SMCL[4] module `MESSY_MAIN_IMPORT_GRID`
4. The SMCL module `MESSY_MAIN_IMPORT_GRID_PAR`

Figure 2.1 illustrates the dependencies of IMPORT_GRID and GRID submodels.

1. The module `MESSY_MAIN_IMPORT_GRID_BI` provides the interface. The subroutines are called from the MESSy entry points in CONTROL via `MESSY_MAIN_IMPORT_BI`. The following MESSy entry points are used by `MESSY_MAIN_IMPORT_GRID_BI`:
   - `IMPORT_GRID_INITIALIZE`: Here the `&RGTEVENTS` namelist is read and the *action string* of each *regrid event* is analysed calling the subroutine `PARSE_STR`[5].

[2] according to the MESSy conventions filenames and module names are identical.
[3] BaseModel Interface Layer
[4] SubModel Core Layer
[5] In subroutine `PARSE_STR` the *action string* is cracked into its components. This subroutine is located in `messy_main_import_grid.f90`

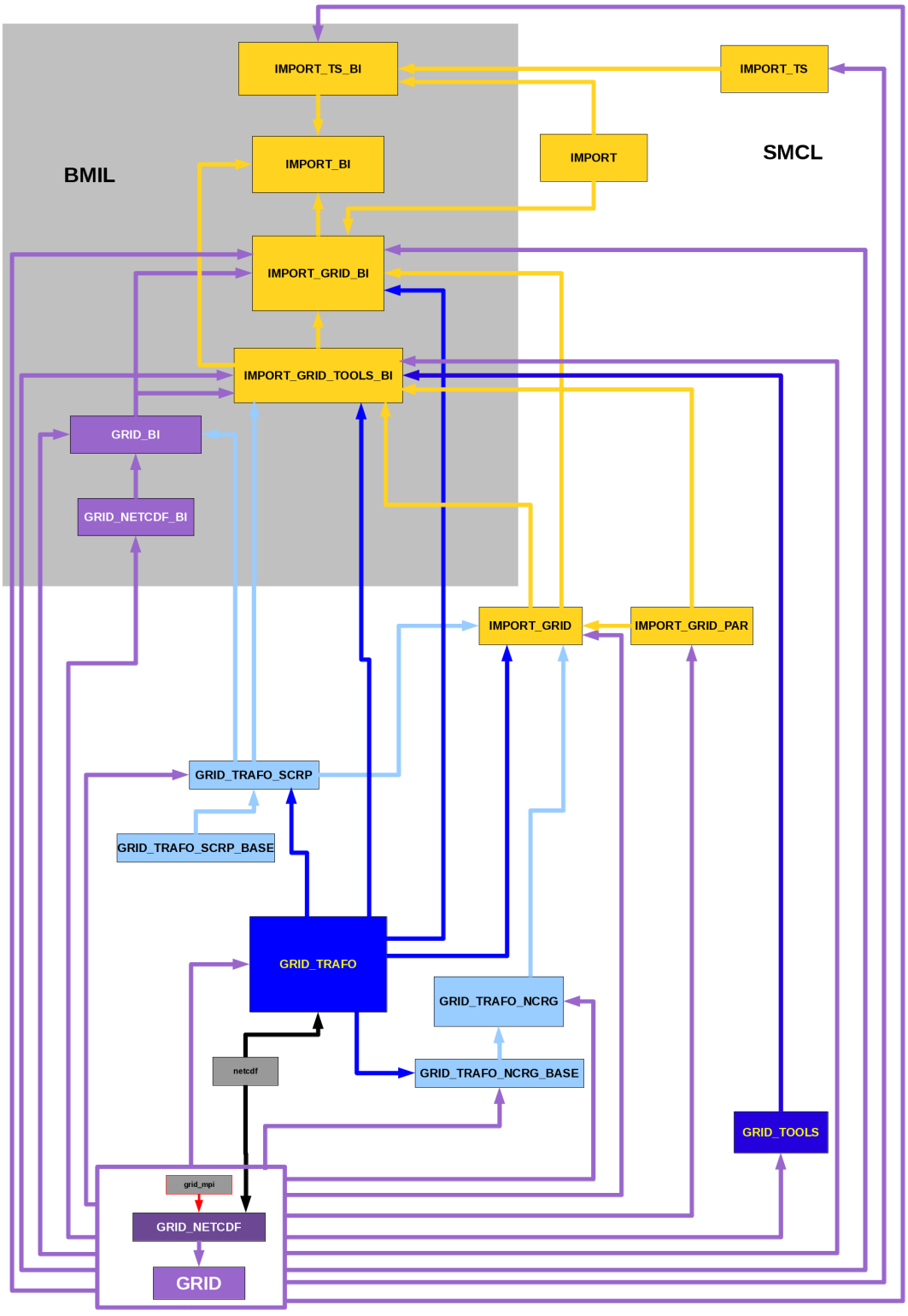

Figure 2.1: Use structure of the generic MESSy submodels GRID (bluish colours) and IMPORT (yellow colours). The Basemodel Interface layer (BMIL) is indicated by grey background colour. Each box equals one module. Arrows point in the direction of the uses, e.g. the submodel core layer (SMCL) modules are used in the BMIL.

- `IMPORT_GRID_INIT_MEMORY`: The dimensions of the regridded data are calculated. For each variable imported to the model a channel object is defined. If attributes are associated with the imported data, these attributes are transformed to the corresponding channel object attributes with the subroutine `ADD_VAR_ATTRIBUTE`.
- `IMPORT_GRID_GLOBAL_START`: During the time loop the `time_events` of all *regrid events* are evaluated and, if required, new input data is processed.
- `IMPORT_GRID_FREE_MEMORY`: At the end of the model simulation internal memory, i.e. memory allocated in addition to the channel objects, is released.

2. The module `MESSY_MAIN_IMPORT_GRID_TOOLS_BI` contains the variable declarations and routines for the `time_event` handling. Additionally, it manages the writing / reading of the ASCII restart file[6] required to store the current status of the *counters*. Furthermore, it contains the interface routines which control the import and regrid process on the basemodel interface level. Two subroutines exist, one for the import of one specific variable (`RGTOOL_BI_READ_NCVAR`) and one for the import of all variables hosted in one file (`RGTOOL_BI_READ_NCFILE`).

3. The SMCL module `MESSY_MAIN_IMPORT_GRID` comprises the routines for the handling of the *counters* and the routines, which control the import and regrid process on the SMCL (`RGTOOL_READ_NCVAR` and `RGTOOL_READ_NCFILE`, respectively). These routines provide the interfaces to GRID_TRAFO and call the respective regridding subroutines. Furthermore, `MESSY_MAIN_IMPORT_GRID` contains the subroutine for managing the data import of the *regrid events* (`READ_CONTROL`), which itself requires the subroutines for reading the namelist (`READ_NAMELIST`), and for parsing the information about the variables (`PARSE_VARSTR`).

4. The SMCL module `MESSY_MAIN_IMPORT_GRID_PAR` contains those routines required for the parallelisation of the regridding process. For SCRIP a domain decomposition parallelisation is possible, i.e., each PE regrids only its subdomain. In contrast to this, NREGRID traditionally (mainly because of the grid domain decomposition applied in ECHAM5) is parallelised over the different variables (see Sect. 2.6.3.4).

Table 2.2 lists the routines of IMPORT_GRID along with a short description. Additionally, it states the module hosting the routine and the section in this manual providing more detailed information on the respective routine or type.

2.6.3.1 RGTEVENT handling

In applications of IMPORT_GRID in interface mode, it is useful to trigger data processing (i.e., reading / regridding of an input field) at specific time steps (`time_events`). For each of these time steps, a specific time slice of the imported variable is used (*counter*). For example, an input file contains monthly emission data, for the years 1995 to 2003 and is used for a simulation of the year 2000. The monthly update of the channel object is triggered by a `time_event`, while the access to the correct data for the year 2000 is managed by the *counter*. Both information are part of the structure `T_RGTEVENT`, which additionally includes the *action string*, thus describing a *regrid event*. The routines required for the `time_event` handling and the overall administration of the `RGTEVENTs` are described in the following, while the *counter* handling is described below (Sect. 2.6.3.2).

[6] "restart" is here used for a user defined checkpointing.

Table 2.2: List of routines and type declarations in IMPORT_GRID. Routines are colored blue and structures red. The file numbers indicate in which module the routines are located: 1: MESSY_MAIN_IMPORT_GRID_BI; 2: MESSY_MAIN_IMPORT_GRID_TOOLS_BI; 3: MESSY_MAIN_IMPORT_GRID; 4: MESSY_MAIN_IMPORT_GRID_PAR.

# Routine name	Short description	file	Sect.
Interface utility routines			2.6.3
PARSE_STR	parsing the action string of RG_TRIG	3	2.6.3
ADD_VAR_ATTRIBUTES	adding the attributes defined during the import to the resulting channel objects	1	2.6.3
RGTEVENT handling			2.6.3.1
TYPE t_rgtevent	structure for the namelist variable RG_TRIG	2	2.6.3.1.1
RGTEVENT_INIT_NML	namelist reading and event initialisation	2	2.6.3.1.2
RGTEVENT_STATUS	inquiring status of time_event and updating *counter*	2	2.6.3.1.3
RGTEVENT_INIT	initialisation of the time_events required for each *regrid event*	2	2.6.3.1.4
RGTEVENT_STAT	evaluation of the status of time_event, called from RGTEVENT_STATUS	2	2.6.3.1.5
RGTEVENT_INDEX	searching for the index of a specific regrid event	2	2.6.3.1.6
***Counter* handling**			2.6.3.2
TYPE t_ncrgcnt	structure for the *counter*	3	2.6.3.2.1
NEW_NCRGCNT	creation of a new list entry in the *counter* list	3	2.6.3.2.2
LOC_NCRGCNT	location of a specific *counter* in *counter* list	3	2.6.3.2.3
GET_NEXT_NCRGCNT	next element in *counter* list	3	2.6.3.2.4
RGTOOL_NCRGCNT_RST	interface for *counter* handling at start and restart	3	2.6.3.2.5
NCRGCNT_HALT	error flag handling	3	2.6.3.2.6
ncrgcnt_error_str	production of an error messages	3	2.6.3.2.7
CLEAN_NCRGCNT_LIST	removing all *counter* list entries	3	2.6.3.2.8
WRITE_NCRGCNT_LIST	dumping *counter* list to ASCII file	3	2.6.3.2.9
READ_NCRGCNT_LIST	reading list of *counters* from ASCII file	3	2.6.3.2.10
MAIN_IMPORT_GRID_WRITE_RESTART	calling WRITE_NCRGCNT_LIST in case of an interruption of simulation	2	2.6.3.2.11
MAIN_IMPORT_GRID_READ_RESTART	calling READ_NCRGCNT_LIST in case of a resumed simulation	2	2.6.3.2.12
IMPORT_GRID_TOOLS_FREE_MEMORY	releasing memory of *counter* list	2	2.6.3.2.13
The importing routines			2.6.3.3
READ_NAMELIST	reading the regridding namelist	3	2.6.3.3.5
PARSE_VARSTR	parsing the variable string (var=...)	3	2.6.3.3.6
RGTOOL_READ_NCVAR	import and regridding of a single variables	3	2.6.3.3.2
RGTOOL_READ_NCFILE	import and regridding of all variables of one data file	3	2.6.3.3.3

Table 2.2: List of routines in IMPORT_GRID (... continued)

# Routine name	Short description	file	Sect.
READ_CONTROL	data import by reading the regridding namelists and afterwards importing the data from the file	3	2.6.3.3.4
Parallelisation			2.6.3.4
t_mpi_def	information of parallel environment	4	2.6.3.4
DISTRIBUTE_VARS_ON_PES	parallelisation along variable space	4	2.6.3.4.1
INIT_PNCREGRID	initialisation of variables used for parallelisation	4	2.6.3.4.2
INIT_PARALLEL	collecting information of parallel environment	4	2.6.3.4.3
EXPAND_FILENAME	expanding the output filename by PE number in case of parallelisation	4	2.6.3.4.4

2.6.3.1.1 TYPE T_RGTEVENT

For the handling of the *regrid events* the type `T_RGTEVENT` hosting the event information is defined:

```
! USED FOR NAMELIST-INPUT
TYPE T_RGTEVENT_IO
   TYPE(io_time_event)         ::  &
                 evt = io_time_event(1, TIME_INC_MONTHS,TRIG_EXACT,0)
   TYPE(T_NCRGCNT)             :: cnt = T_NCRGCNT('', -1, -1, -1, -1)
   CHARACTER(LEN=RGTMAXACTSTR) :: act = ''
END TYPE T_RGTEVENT_IO

! INTERNAL WORKSPACE
TYPE T_RGTEVENT
   TYPE(T_RGTEVENT_IO) :: io
   TYPE(time_event)    :: event
END TYPE T_RGTEVENT
```

Type `T_RGTEVENT` consists of the corresponding structure component of type `T_RGTEVENT_IO` used for the namelist input of the *regrid event*, and an internally used `time_event` (see TIMER User Manual). The type `T_RGTEVENT_IO` contains the structure components described above: the `io_time_event`, the *counter* (type `T_NCRGCNT`) and the *action string*. For the definition of the *counter* see Sect. 2.6.3.2.

2.6.3.1.2 SUBROUTINE RGTEVENT_INIT_NML

The `&RGTEVENTS` namelist is read in this subroutine by calling the private subroutine `RGTEVENT_READ_NML_CPL` on the I/O-PE and afterwards broadcasted to all PEs. Additionally, the individual `time_events` are initialised via subroutine `RGTEVENT_INIT`, followed directly by the investigation of the *regrid event* status by calling the subroutine `RGTEVENT_STATUS`.

2.6.3.1.3 SUBROUTINE RGTEVENT_STATUS

The interface of the subroutine `RGTEVENT_STATUS` is defined by

```
! ------------------------------------------------------------------
SUBROUTINE RGTEVENT_STATUS(status, cpos, rgt, modstr, name, index, action &
                         , lstop, linit)

  ! RETURNS THE EVENT STATUS (flag) OF A NAMED (name)
  ! OR INDEXED (index) RGT-EVENT IN A LIST (rgt)
  !
  ! Author: Patrick Joeckel, MPICH, Mainz, October 2002

  ...

  IMPLICIT NONE
  ...

  ! I/O
  LOGICAL,                      INTENT(OUT)            :: status ! event status
  INTEGER,                      INTENT(OUT)            :: cpos   ! counter pos.
  TYPE(T_RGTEVENT), DIMENSION(:), POINTER              :: rgt    ! RGT-event list
  CHARACTER(LEN=*),             INTENT(IN)             :: modstr ! calling module
  CHARACTER(LEN=*),             INTENT(IN),  OPTIONAL  :: name   ! name of event
  INTEGER,                      INTENT(IN),  OPTIONAL  :: index  ! index of event
  CHARACTER(LEN=RGTMAXACTSTR),  INTENT(OUT), OPTIONAL  :: action ! action string
  LOGICAL,                      INTENT(IN),  OPTIONAL  :: lstop  ! stop on error
  LOGICAL,                      INTENT(IN),  OPTIONAL  :: linit  ! initialize?
! ------------------------------------------------------------------
```

The status of the `time_event` is calculated using the subroutine `RGTEVENT_STAT`. Secondly, the *counter* of the *regrid event* is updated using subroutine `RGTOOL_NCRGCNT_RST`. Last but not least, the output parameters, i.e., the current position of the *counter* (`cpos`) and the `status` flag are set. If the *action string* is requested, it is copied to the optional parameter `action`.

For identification of the respective *regrid event*, the `name` or the `index` are required. If both are not present the returned status is non-zero. If only the `name` is provided, the `index` is obtained by calling the subroutine `RGTEVENT_INDEX`. If no *regrid event* with the provided `name` is found, the returned status flag is non-zero. The same happens, if the given index is not available in the list of *regrid event*s. Note: if the `index` is provided during the subroutine call, the value of `name` is ignored. There is no internal test, if `name` and `index` fit.

2.6.3.1.4 SUBROUTINE RGTEVENT_INIT

This subroutine performs the initialisation of the `time_event`s corresponding to the *regrid events*.

2.6.3.1.5 SUBROUTINE RGTEVENT_STAT

This subroutine evaluates the status of a `time_event` by calling the TIMER function `event_state`.

2.6.3.1.6 SUBROUTINE RGTEVENT_INDEX

This subroutine searches the list of *regrid events* to locate the `index` of a *regrid event* of a specific name.

2.6.3.2 *Counter* handling

This subsection describes how the *counter* information is processed. For this, `MESSY_MAIN_IMPORT_GRID` contains an additional type declaration, and the corresponding subroutines to provide this functionality.

2.6.3.2.1 TYPE T_NCRGCNT

The type `T_NCRGCNT` provides a structure to store *counter* information:

```
TYPE T_NCRGCNT
  CHARACTER(LEN=NCCNTMAXNLEN) :: name    = ''
  INTEGER                     :: start   = 1
  INTEGER                     :: step    = 1
  INTEGER                     :: reset   = 1
  INTEGER                     :: current = 1
END TYPE T_NCRGCNT
```

`name` is used to identify the *counter* (e.g., in a list of *counters*), `start` contains the minimum *counter* position, `step` the increment, and `reset` the maximum *counter* position. The current *counter* position can for instance be used to read / regrid a specific time step of a netCDF variable by setting the parameter `t` of subroutine `RGTOOL_READ_NCVAR` or subroutine `RGTOOL_READ_NCFILE` to `current` (see Sect. 2.6.3.3).

To process a large number of *counters* the individual *counters* are stored in a concatenated list:

```
  TYPE T_NCRGCNT_LIST
     CHARACTER(NCCRSTRL)           :: mname  = ''
     TYPE(T_NCRGCNT)               :: this
     TYPE(T_NCRGCNT_LIST), POINTER :: next => NULL()
  END TYPE T_NCRGCNT_LIST
```

`mname` is the name of the submodel defining this *counter*.

2.6.3.2.2 SUBROUTINE NEW_NCRGCNT

With this subroutine a new list entry in the concatenated list of *counters* is created.

```
! ------------------------------------------------------------
  SUBROUTINE new_ncrgcnt(status, mname, cnt)

    IMPLICIT NONE

    ! I/O
    INTEGER,          INTENT(OUT)  :: status
    CHARACTER(LEN=*), INTENT(IN)   :: mname
    TYPE(t_ncrgcnt),  INTENT(IN)   :: cnt
! ------------------------------------------------------------
```

Input to the subroutine is a *counter* of type `T_NCRGCNT` and the name `mname` of the submodel defining the *counter*, i.e., `'import_grid'`. (This option is a relict of earlier versions without unique interface for data import into a MESSy model: all submodels importing data had to include there own interface to NCREGRID. In this case it was important to know, which submodel defined the respective *counter*). A `status` flag is handed back to the calling routine providing information about errors or the information, if a *counter* already exists.

2.6.3.2.3 SUBROUTINE LOC_NCRGCNT

This subroutine locates a *counter* in the concatenated list:

```
! ------------------------------------------------------------------
SUBROUTINE loc_ncrgcnt(status, mname, cname, cntptr)

  IMPLICIT NONE

  ! I/O
  INTEGER,              INTENT(OUT) :: status
  CHARACTER(LEN=*),     INTENT(IN)  :: mname
  CHARACTER(LEN=*),     INTENT(IN)  :: cname
  TYPE(t_ncrgcnt),      POINTER     :: cntptr
! ------------------------------------------------------------------
```

The name of the defining submodel (`mname`) and the *counter* name (`cname`) are input to the routine. If the *counter* is found, the pointer `cntptr` is associated to the *counter* and the `status` is zero. Otherwise, the pointer `cntptr` is NULLIF(Y)ied and the `status` flag provides information about the reason:

status	error
5003	`mname` too long
5006	`cname` too long
5005	*counter* does not exist

2.6.3.2.4 SUBROUTINE GET_NEXT_NCRGCNT

The subroutine `GET_NEXT_NCRGCNT` can be used to loop over all currently stored *counters* in the internal list, e.g., to search for a specific *counter*:

```
LOGICAL                  :: last
TYPE(t_ncrgcnt), POINTER :: cptr
  ...
DO
  CALL GET_NEXT_NCRGCNT(last, cptr)
  IF (last) EXIT
  ! check cptr here
  ....
END DO
```

`last` is `.TRUE.`, if the end of the list is reached. This construction is required, since the number of *counters* stored in the internal list is not known a priori. Internally the subroutine distinguishes two modes: `MODE_INIT` and `MODE_CONT`. At the first call, the `MODE` is set to `MODE_INIT`. This triggers

that the internal pointer is set to the start element of the concatenated list of *counters* (`GRGTLIST`). Afterwards the `MODE` is set to `MODE_CONT`. If the first list element exists, this pointer is returned and `MODE` is still `MODE_CONT`, so that at the next call of the subroutine the next pointer in the list can be accessed.

```
! ------------------------------------------------------------------
  SUBROUTINE GET_NEXT_NCRGCNT(last, cntptr)

    IMPLICIT NONE

    ! I/O
    LOGICAL,              INTENT(OUT) :: last
    TYPE(t_ncrgcnt),      POINTER     :: cntptr
! ------------------------------------------------------------------
```

As long as more list elements exist, upon return, `cntptr` is associated to the next element and `last` is set `.FALSE.`. If no more elements are available, `last` is set `.TRUE.` and `cntptr` keeps its association. Additionally `MODE` is reset to `MODE_INIT`, indicating that at the next call of the subroutine `GET_NEXT_NCRGCNT` the cycling of the concatenated list must start at the very beginning of the list.

2.6.3.2.5 SUBROUTINE RGTOOL_NCRGCNT_RST

This subroutine provides an interface for the automatic handling of *counter* information, including

- the update of the *counter* (triggered by `event==.TRUE.`): The actual *counter* position `current` is incremented by `step`. If the result is larger than `reset`, `current` is reset to `start`. This allows cyclic counting.
- the unambiguous storage of a specific *counter* in the concatenated list of all *counters* (triggered by `linit==.TRUE.`). This makes the *counter* accessible from everywhere, which is required to save the *counters* in an external file at model interruption during a user defined checkpointing (restart).
- the continuous update of the actual *counter* and its copy stored in the concatenated central list.

The subroutine is called with the following parameters:

```
! ------------------------------------------------------------------
  SUBROUTINE RGTOOL_NCRGCNT_RST(mname, start, restart, event, c, lout, linit)

    ! MANAGES I/O OF COUNTER INFORMATION AT START AND RESTART
    !
    ...

    IMPLICIT NONE

    INTRINSIC :: PRESENT, TRIM

    ! I/O
```

```
    CHARACTER(LEN=*), INTENT(IN)    :: mname
    LOGICAL,          INTENT(IN)    :: start   ! .true. at first time step
    LOGICAL,          INTENT(IN)    :: restart ! .true. at first time step of
                                                ! rerun
    LOGICAL,          INTENT(IN)    :: event   ! .true. on event
    TYPE(T_NCRGCNT),  INTENT(INOUT) :: c       ! counter - struct
    LOGICAL,          INTENT(IN)    :: lout
    LOGICAL,OPTIONAL, INTENT(IN)    :: linit

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

- `mname` is a name identifying the defining submodel.
- `start` must be `.TRUE.` only at the very first model time step. It is used to prevent a *counter* update immediately at model start, if at the same time the event is triggered.
- `restart` must be `.TRUE.` only after the first initialisation of the *counter* list from the restart file. This is used to restore the actual *counter* from the list, even if the event is not triggered.
- `event` is `.TRUE.`, indicating that the *counter* update is triggered.
- `c` is the *counter* structure of the actual *counter*.
- `lout` is `.TRUE.` for diagnostic output.
- `linit` is an optional switch (default: `.FALSE.`). If set to `.TRUE.`, it prevents *counters* of the same name and `mname` to be updated, if the *counter* exists already in the list, but creates a new entry in the *counter* list, if it is new. This is required to enable the definition of additional *regrid events* after restart.

The sequence of operations is as follows:

1. If `start`, `restart`, and `event` are all `.FALSE.`, the subroutine is left immediately.
2. The algorithm locates the specified *counter* `c` by its name and `mname`. If the *counter* is found, the content of the respective *counter* in the list is copied to the actual *counter* `c`.
3. The initialisation flag `linit` is checked: Only if `linit` is `.TRUE.` the presence of the *counter* in the list is re-evaluated: If it exists in the list, the routine returns with a non-zero error status, since during the initialisation, the *counter* must not yet exist. If the *counter* is new, however, it is added to the list.
4. The algorithm checks, if an update has been triggered. Only if `event` is `.TRUE.` and `start` is `.FALSE.`, an existing actual *counter* and its copy in the central list are updated, as described above. If the *counter* does not exist in the list, the routine exits with a non-zero status.

2.6.3.2.6 SUBROUTINE NCRGCNT_HALT

This routine provides the handling of the error status flags delivered by other subroutines. It uses the RMSG subroutine from GRID to stop the simulation, if an error occurs.

```
! ------------------------------------------------------------------
 SUBROUTINE ncrgcnt_halt(substr, status)

   ! MESSy
   USE messy_main_constants_mem,  ONLY: STRLEN_VLONG
   USE messy_main_grid_netcdf,    ONLY: RGMSG, RGMLE

   IMPLICIT NONE
   ! I/O
   CHARACTER(LEN=*), INTENT(IN)  :: substr
   INTEGER,          INTENT(IN)  :: status
! ------------------------------------------------------------------
```

Input to the subroutine are a character string (`substr`) indicating the calling routine and the `status` flag to be evaluated.

2.6.3.2.7 FUNCTION ncrgcnt_error_str

This function provides a meaningful error string associated to the respective status flag. It is used by subroutine `NCRGCNT_HALT`.

2.6.3.2.8 SUBROUTINE CLEAN_NCRGCNT_LIST

This subroutine is used to remove all *counter* list entries, i.e., to clean up the memory.

2.6.3.2.9 SUBROUTINE WRITE_NCRGCNT_LIST

With the subroutine `WRITE_NCRGCNT_LIST` the complete concatenated list of currently stored *counter* information is dumped into an ASCII file. This is required for restarts.

2.6.3.2.10 SUBROUTINE READ_NCRGCNT_LIST

With subroutine `READ_CRGCNT_LIST`, the *counter* information is restored from a file previously written by subroutine `WRITE_NCRGCNT_LIST`. This is required for restarts.

2.6.3.2.11 SUBROUTINE MAIN_IMPORT_GRID_WRITE_RESTART

This BMIL subroutine calls the SMCL routine `WRITE_NCRGCNT_LIST`, if writing of restart files is triggered.

2.6.3.2.12 SUBROUTINE MAIN_IMPORT_GRID_READ_RESTART

This BMIL subroutine calls the SMCL routine `READ_NCRGCNT_LIST`, if a simulation is resumed after restart.

2.6.3.2.13 SUBROUTINE IMPORT_GRID_TOOLS_FREE_MEMORY

This BMIL subroutine initiates the initialisation of the concatenated list of *counters* by calling the SMCL routine `CLEAN_NCRGCNT_LIST`.

2.6.3.3 Interfaces for file reading and regridding

The SMCL file `MESSY_MAIN_IMPORT_GRID` contains all subroutines managing the data import and regridding. The subroutines `RGTOOL_READ_NCVAR` and `RGTOOL_READ_NCFILE` are the drivers of these processes for import of single variables (`NCVAR`), or all variables contained in one file (`NCFILE`), respectively. The corresponding BMIL interfaces are the subroutines `RGTOOL_BI_READ_NCVAR` and `RGTOOL_BI_READ_NCFILE`, respectively, located in `MESSY_MAIN_IMPORT_GRID_TOOLS_BI`.

2.6.3.3.1 Subroutines RGTOOL_BI_READ_NCVAR and RGTOOL_BI_READ_NCFILE

These two subroutines provide the main entry points on the BMIL of IMPORT_GRID and internally call the corresponding SMCL routines, which acutally do the reading and regridding of the input data and the main IMPORT_GRID BMIL. After processing the data, the format of the data is converted from the internal 1D vector to the 4D arrays corresponding to the respective representation. This conversion is performed by the GRID subroutine `RGTOOL_CONVERT`. Here especially the order of the ranks in the 4D arrays is important. Additionally, if the processing is not performed in parallel mode and a new grid has been created by the SMCL routines, this grid is broadcasted to all PEs. In case of parallel domain decomposed input (`ldompar =.TRUE.`) each PE hosts its own sub-grid definition.

2.6.3.3.2 SUBROUTINE RGTOOL_READ_NCVAR

This subroutine manages the import and the regridding of the data for single variables. The interface is given by:

```
SUBROUTINE RGTOOL_READ_NCVAR(status, iou, nmlfile, vname, t, var   &
                           , iipol, ogridid, igridid, SCRIP_ID    &
                           , lrg, lrgx, lrgy, lrgz, lok , oarea   &
                           , convgrid, ldompar, lvarpar)
```

The meaning of the individual arguments is briefly described in Table 2.3. The subroutine starts with the initialisation of local variables controlled by the optional arguments. First the output grid structures (`conv_grid` and `ogrid`) are initialised. These two grids are different, if the subroutine `RGTOOL_READ_NCVAR` is used to simply read the input field. Additionally, the structure `var` containing the read or regridded field is initialised. As in principle, the namelist file of one *regrid event* can contain any number of `&RGTEVENT` namelists, an endless DO-loop builds the main part of the subroutine. To control the cycling conditions three internal parameters are used:

- `RG_CTRL`:
  - `RG_SCAN`: scan the namelist
  - `RG_PROC`: process data
  - `RG_STOP`: stop data processing and namelist scanning
- `RG_NML`:
  - `NML_NEXT`: read next namelist
  - `NML_STAY`: read namelist again and import data
- `RG_STATUS`:
  - `RGSTAT_START`: start regridding procedure

- RGSTAT_CONT: continue endless DO-loop
- RGSTAT_STOP: exit endless DO-loop

Start values at the beginning of the endless loop are RG_CTRL=RG_SCAN, as the namelist file has to be scanned for the required namelist entry, and RG_NML=RG_NEXT as the next namelist should be read. Figure 2.2 illustrates the flow of the subroutine RGTOOL_READ_NCVAR. First in the endless DO-loop, the subroutine READ_CONTROL is called. In that subroutine, depending on the switches RG_CTRL and RG_NML, a namelist and the required data are read (see Sect. 2.6.3.3.4). After exiting the subroutine READ_CONTROL, it is checked whether the currently read namelist entry (variable name) is the requested variable. If not, RG_CTRL and RG_NML are set to RG_SCAN and RG_NEXT, respectively, and the endless DO-loop cycle starts again. If the required variable has been found, the data is processed according to llrg:

- llrg=.FALSE.: In the initialisation phase of the model, i.e., for dimensioning the output data (channel objects), the namelists are read once, without starting the remapping. Thus, llrg is set .FALSE. in the initialisation phase. In this case, only raw data input is required. RG_CTRL is set to RG_SCAN and the imported raw data (local structure rvar) is copied to the output structure var. Additionally, the input grid specifications are added to the list of geo-hybrid grids by calling the GRID subroutine NEW_GEOHYBGRID. Furthermore, if the data set should be regridded by SCRIP, the SCRIP interpolation weights are calculated and stored in the respective data structures. Last but not least, the input grid is copied to the output structure conv_grid for the dimensioning of the output structures.

variable	INTENT	type		meaning
status	OUT	INTEGER		status flag for GRID routines (0 = ok, if /= 0 information is provided via function grid_error)
iou	IN	INTEGER		I/O unit
nmlfile	IN	CHARACTER		namelist filename
vname	IN	CHARACTER		name of the variable to process
t	IN	INTEGER		times step in netCDF file
var	OUT	TYPE(t_ncvar)		out structure
iipol	IN	INTEGER		interpolation method (one of GTRF_NONE, GTRF_SCRP or GTRF_NRGT)
ogridid	IN	INTEGER		ID of destination grid
igridid	INOUT	INTEGER		ID of source (netCDF file) grid
SCRIP_ID	INOUT	INTEGER		ID of SCRIP structure
lrg	IN	LOGICAL	OPTIONAL	flag for interpolation. If .FALSE. data is only read in.
lrgx	IN	LOGICAL	OPTIONAL	flag for interpolation in first horizontal (x-)direction
lrgy	IN	LOGICAL	OPTIONAL	flag for interpolation in second horizontal (y-)direction
lrgz	IN	LOGICAL	OPTIONAL	flag for vertical interpolation
lok	OUT	LOGICAL	OPTIONAL	success flag for subroutine (lok = .TRUE. includes status == 0, but not the other way round.)
oarea	IN	REAL(dp)	OPTIONAL	area of zells in output grid, might be used in SCRIP
conv_grid	INOUT	TYPE(t_geohybgrid)	OPTIONAL	grid structure, which is required for grid conversions in the interface subroutine RGTOOL_BI_READ_NCVAR
ldompar	IN	LOGICAL	OPTIONAL	parallelisation over the destination domain
lvarpar	IN	LOGICAL	OPTIONAL	parallelisation over variables

Table 2.3: List of arguments of subroutine RGTOOL_READ_NCVAR

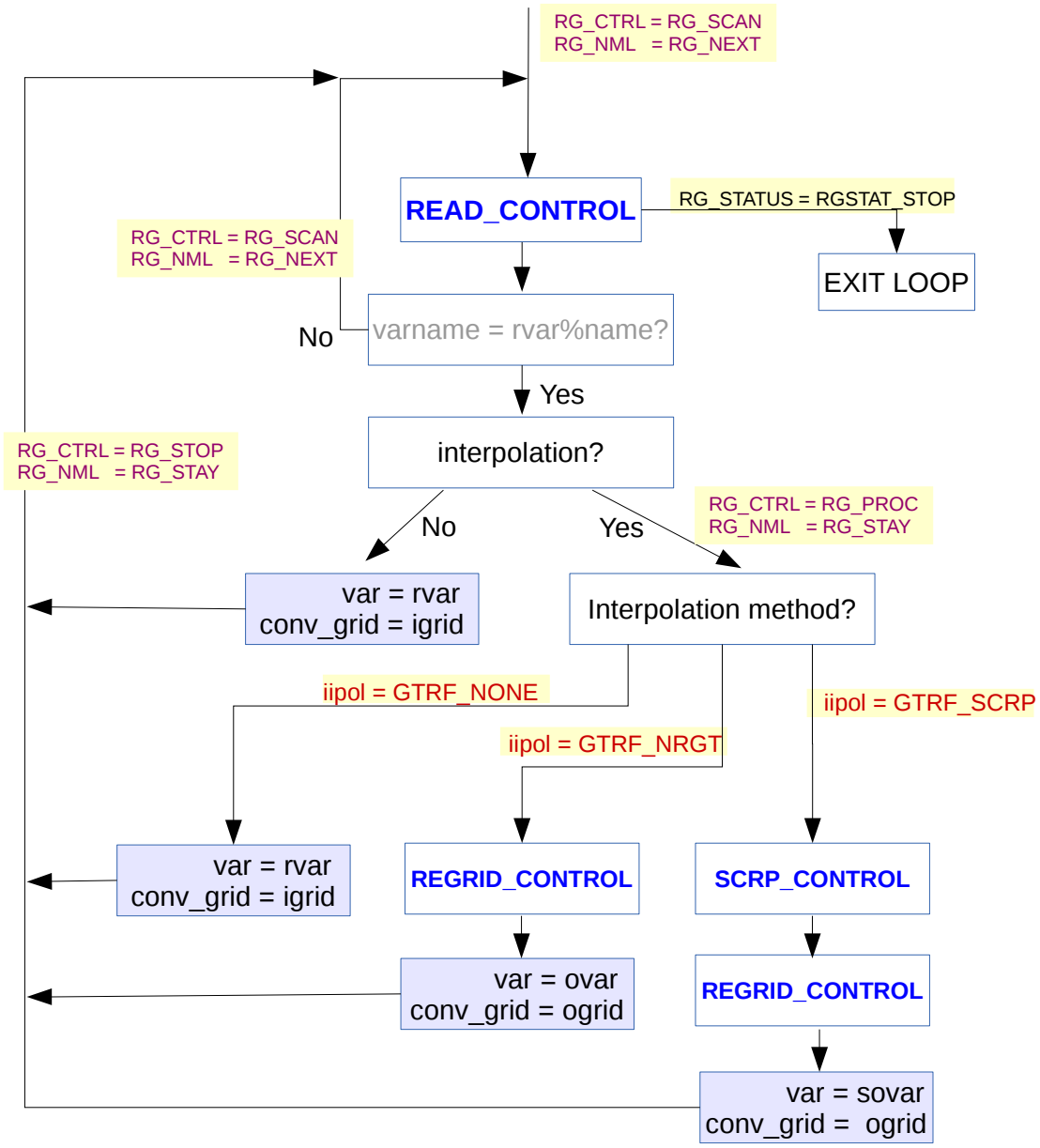

Figure 2.2: Flux diagram for endless DO-loop in subroutine `RGTOOL_READ_NCVAR`. Blue letters indicate subroutine calls, the light yellow boxes show settings of the internal switches.

- `llrg=.TRUE.:` This signifies (usually during the time loop) that regridding of the imported raw data is requested. In this case, `RG_CTRL` is set to `RG_PROC`. Depending on the chosen method, the data is regridded:
  - `GTRF_NONE`: If no grid transformation is requested, the raw data is copied to the output structure `var` and the input grid is copied to the output grid structure `conv_grid`.

- GTRF_NRGT: If regridding by NREGRID is chosen, REGRID_CONTROL is called (see GRID-User-Manual). Afterwards, the regridded structure ovar is copied to the output structure var and the destination grid is copied to conv_grid.
- GTRF_SCRP: This is the most complex call, as SCRIP transforms only horizontal grids. Thus, first it is checked, if horizontal regridding is requested (llrgx=.TRUE. and / or llrgy=.TRUE.). In this case, SCRIP_CONTROL is called, otherwise the raw data is copied to the intermediate structure sovar. If vertical grid transformation is requested, NREGRID is used. In this case, the grid specification has to be adapted to the horizontally regridded data. This is especially important for data on curvi-linear grids, as NREGRID can not deal with this. As NREGRID is only used for the vertical grid transformation, the horizontal grid is defined in a way, that NREGRID can handle it. A simple example: For the COSMO model grid, the coordinates of the horizontal grid are defined in the rotated coordinate system. This grid is "pseudo-geographically-rectangular" and not curvi-linear and NREGRID can deal with it. After redefining the input grid, REGRID_CONTROL is called with the arguments llrgx=.FALSE. and llrgy=.FALSE., as the horizonal mapping was already performed by SCRIP. After the vertical grid transformation, the output structures var and conv_grid are filled.

After processing, RG_CTRL is set to RG_STOP and some local structures and variables are reinitialised to free the memory. In the next call of READ_CONTROL some further internally allocated memory is released and the endless DO-loop is exited.

2.6.3.3.3 SUBROUTINE RGTOOL_READ_NCFILE

The systematic of this subroutine is the same as for RGTOOL_READ_NCVAR. In contrast to the subroutine RGTOOL_READ_NCVAR, where the variable name (vname) is parameter of the subroutine, in the subroutine RGTOOL_READ_NCFILE a filename (fname) is provided and the namelist file is searched for the respective filename until this file is found. Afterwards, all variables named in the respective namelist are processed.

2.6.3.3.4 SUBROUTINE READ_CONTROL

This subroutine is the "heart" of the actual data import. It reads the individual &regrid namelists and inputs the raw data fields accordingly. READ_CONTROL is split in two parts: the reading and analysis of the namelist (subroutine READ_CONTROL_INIT) and the actual import of the data (subroutine READ_CONTROL_WORK). The work flow of READ_CONTROL is determined by the local switches GSTAT, GCTRL and GNML, which are the local counterparts to the switches RG_STATUS, RG_CTRL and RG_NML used in the subroutines RGTOOL_READ_NCVAR and RGTOOL_READ_NCFILE.

In READ_CONTROL_INIT, depending on the control switches, different procedures are passed through:

- GSTAT == RG_START .OR. GNML == NML_NEXT:
  In this case, the &regrid namelist is read on all PEs (more details about the parallelisation are given in Subsect. 2.6.3.4) by the subroutine READ_NAMELIST as described in Sect. 2.5. The string of namelist entry var, specifying the names of the variables and optional scaling factors, is parsed using the subroutine parse_varstr. If an error in reading the namelist occurs, or the end of the namelist file is reached, GSTAT is set to RGSTAT_STOP. After reading the namelist, the variables controlling the parallelisation of the data processing (see Sect. 2.6.3.4) are set: If ldompar=.TRUE., i_am_worker is always .TRUE. and nproc_work =1. In case of lvarpar=.TRUE., the subroutine DISTRIBUTE_VARS_ON_PES distributes the variables

on the available PEs and sets the switches `i_am_worker` and `nproc_work` accordingly (see Sect. 2.6.3.4.1). If no parallelisation is requested (i.e., `ldompar=.FALSE.` and `lvarpar=.FALSE.`) the subroutine `READ_CONTROL` has to be called on the I/O-PE only. For this PE the switches `i_am_worker` and `nproc_work` are set to `.TRUE.` and `1`, respectively.

- `GNML == NML_STAY`:
  In this case the namelist is not read again, but the variable string is parsed again by subroutine `parse_varstr`.

If at this point `GSTAT = RGSTAT_STOP`, the I/O-unit is closed and the memory of various local variables is released.

The second subroutine, `READ_CONTROL_WORK`, is only called, if `i_am_worker = .TRUE.`.

- At the beginning the respective step selector, driving the input of the correct time step from a multi-time step file (according to the *counter*) are checked and set. Secondly, in case of parallelisation, the filename of the output file, if direct output in a netCDF file is requested, needs to be expanded by the processor number, as otherwise different PEs would write differently dimensioned data to the same file via serial netCDF output.
- After the initialisation is finished, the import starts with reading of the input grid (`CALL IMPORT_GEOHYBGRID(gi)`). If the output grid (`gg`) is provided by a file and not online, it is also read. Both grids are successively checked (`CALL CHECK_GEOHYBGRID`), whether they provide sufficient information.
- If the procedure is run in domain parallel decomposition (`ldompar=.TRUE.`) the input grid is reduced to the part required for the subdomain on the respective PE (`CALL REDUCE_INPUT_GRID`). This step is very important to reduce the memory footprint for the imported raw data, as well as for the whole interpolation process.
- Subsequently, all variables attributed to the respective PE are imported (`CALL IMPORT_NCVAR`). In case of a reduced input grid, the variables `zstart` and `zcount` provide the information, which hyperslice of the variable has to be read (for more detail see the GRID-User-Manual). After import, a check of the grid of the imported variable is performed (`CALL CHECK_NCVAR_ON_GROHYBRID`).
- If everything is ok, the imported variables are copied to the output structures, renamed and scaled, and the time axis is adjusted, if required.
- At the end, locally allocated memory is released and grid specific structures are initialised.

2.6.3.3.5 SUBROUTINE READ_NAMELIST

This subroutine is called by `READ_CONTROL` and reads the `&regrid` namelists. The content of the namelists is described in detail in Sect. 2.5.

2.6.3.3.6 SUBROUTINE PARSE_VARSTR

The subroutine `parse_varstr` parses the string of the `var` specifier in the `&regrid` namelist (see Sect. 2.5). It might look like

```
var = 'NO=NOx_flux,2.00671e+25;LAI:IDX;SO2=SO2_flux,9.39932e+24;',
```

i.e., it contains an arbitrary number of specifications. For each imported variable, a new name can be assigned, a scaling factor and the regridding method can be given. In this example the input field `NOx_flux` is renamed to `NO` and scaled by `2.00671e+25`. The `LAI` is index regridded. The entries for the individual input fields are semicolon seperated, the renaming is indicated by an equal sign and the scaling factor is seperated by a comma, while the integration method is seperated by a colon. This syntax is used to process the string.

- First the string is split in the substrings between the semicolons yieldung `nvar` substrings.
- Afterwards, each substring is searched for the colon and the comma. The content after these signs is then associated to the RGTstr (regrid methods as string) and the scaling factor, respectively.
- Along, if an equal sign is part of the substring, the string left (right) of the equal sign is assigned as destination (source) name of the variable, respectively.
- After all substrings are processed, the acquired information are forwarded to the calling subroutine in form of 1D arrays dimensioned by `nvar`. The arrays are
  - `ivar`: input variable names, i.e., the names of the fields in the file,
  - `ovar`: output variable names, i.e., the names associated by IMPORT_GRID to the respective variables,
  - `scl`: the scaling factor (default: 1.),
  - `RGT`: integer indicating the regridding method (default: `RG_INT`)
  - `RGTstr`: string indicating the regridding method (default: `'INT'`)

2.6.3.4 Parallelisation of GRID_TRAFO

In principle the data import and the regridding can be processed in parallel. Depending on the calling model different methods are applicable. For the stand-alone tools parallelisation is not necessarily required. For 3D models, one option is to utilise the parallel domain decomposition of the basemodel, i.e., each PE processes the data required for its respective subpart of the model domain (`ldompar=.TRUE.`). For IMPORT_GRID this is the case for the COSMO model. In models with a more complex domain decomposition (e.g., ECHAM5) this is not straightforwardly applicable. Thus, either no parallelisation at all is chosen, or, if the number of variables contained in one file is large enough, the parallelisation can be applied over the variables (`lvarpar=.TRUE.`) This is the case for the data import in the tracer initialisation in EMAC[7]. Internally, `ldompar` is always prefered over `lvarpar`, e.g. the tracer initialisation in COSMO/MESSy is always performed in domain decomposition.

The information about which PE is processing what is internally provided by the logical switch `i_am_worker`, which is `.TRUE.` if the PE is supposed to process data, and the number of processing units (`nproc_work`). Additionally, to enable the coordinated work of the PEs, the structure `my_mpi` of type `T_MPI_DEF`

```
TYPE T_MPI_DEF
   INTEGER :: rank  = -1
   INTEGER :: nproc = -1
   INTEGER :: comm  = -1
END TYPE T_MPI_DEF
```

[7]Note: As a special case, tracer initialisation is triggered by the BMIL of TRACER and not by the BMIL of IMPORT.

contains the information about the parallel environment: `comm` is the communicator, `nproc` the number of PEs belonging to `comm` and `rank` is the rank of the respective PE in `comm`. Furthermore, the 1D variable `pe_list` provides a list, which variable is handled on which PE. In case of `lvarpar=.TRUE.` these variables and structure components are set in the subroutine `DISTRIBUTE_VARS_ON_PES`. In all other cases, they are initialised with meaningful values. This happens partly in the `READ_CONTROL` and partly in the helper routines `INIT_PNCREGRID` and `INIT_PARALLEL`.

2.6.3.4.1 SUBROUTINE DISTRIBUTE_VARS_ON_PES

This subroutine is called from `READ_CONTROL`. Dependent on the number of imported variables and the number of PEs, the variables are distributed over the PEs. To reduce the memory load, only a maximal number of working PEs is set. First, the overall number of working PEs (`nproc_work`) is determined as the minimum of

- the number of available PEs,
- the number of variables to import, and
- the maximum number of working PEs.

Afterwards, the variables are distributed among the working PEs. This information is hosted in the variable `pe_list`. The variables are distributed such, that the working PEs have the largest possible distance to each other, which in the end leads to as few as possible variables per node. For example, during a simulation with 256 tals (32 tasks per node), 16 fields should be processed at once. In this case, `pe_list` would be

```
pe_list(0:15) = (0,16,32,48,64,80,96,112,128,144,160,176,192,208,224,240)
```

which means that on each node 2 variables are processed (variable 1 on PE 0, variable 2 on PE 16, variable 3 on PE 32 ... → on node 1 the processes 0 and 16 (i.e. 2 processes) are working).

After the calculation of `pe_list`, it is counted for each PE, how many variables are processed. After the allocation of the local variables, the imported raw data is copied to the local variables and the memory for the original (global) import fields is released.

2.6.3.4.2 SUBROUTINE INIT_PNCREGRID

If no parallelisation over the variable number occurs, the `pe_list` has no meaning. Nevertheless, it needs to be allocated to avoid run time errors. The subroutine `INIT_PNCREGRID` sets a default.

2.6.3.4.3 SUBROUTINE INIT_PARALLEL

With this subroutine the information about the parallel environment is copied to the local structure `my_mpi` of type `T_MPI_DEF`.

```
my_mpi%rank  = p_pe
my_mpi%nproc = p_nprocs
my_mpi%comm  = p_all_comm
```

`my_mpi%rank` contains the number of the corresponding PE. `my_mpi%nprocs` provides the number of overall PEs and `my_mpi%comm` provides the MPI communicator for the `my_mpi%nprocs` PEs

2.6.3.4.4 SUBROUTINE EXPAND_FILENAME

In principle it is possible to request direct output of the processed data. For this an output filename can be given in the namelist. In parallel mode, the PE number has to be added to the output filename, in order to avoid parallel access to the same file of different PEs via serial netCDF output.

Chapter 3

IMPORT_TS

IMPORT_TS provides a unified interface for the reading of abstract time series data, i.e., data available for a specific period in time and with a specific number of parameters.

3.1 Namelist Control*

IMPORT_TS is driven by the `&CTRL_TS` namelist. See Fig. 3.1 for an example. Each `TS` entry describes one time series data set. The meaning of the components is:

- The first string defines the name of the time series data set and thus the name of the CHANNEL object containing the finally processed data. By means of this name the data can be accessed in other parts of the model.
- The second string comprises the name, including the full path, of the data file. Only for netCDF files, additionally the string contains the name of the variable to be read. The variable name has to be given at the beginning of the string and is seperated from the filename by an `@`-sign. In the example in Fig. 3.1 `TS(1)` defines the time series data named `'exnc'`. The variable in the netCDF file is named `"EXNC"`, while the data file is found under `/DATA/exnc/EXNC_1950_2012.nc`.
- The next two float entries determine the valid range of the data. In case of `TS(1)` in Fig. 3.1 this is between -99.9 and 99.9. The default valid range is between `-HUGE(0.)` and `HUGE(0.)`[1].
- The next two integer variables set the valid time range for the time series data, i.e., if data is provided in cases where the simulation date lies outside of the time span covered by the data file. If set to "0" the model execution is stopped, where as "1" allows for the continuation of the simulation. In the second case, the data of the nearest point in time present in the file is used. As the desired policy may differ for dates before and after the covered time span, the first integer determines the method used for dates prior to the time span comprised in the file, and the second integer the method used after the provided time span. In the example (Fig. 3.1) the simulation would be stopped, if a date outside the time frame covered by the exnc file (`TS(1)`) is reached. For `TS(2)`, the simulation will be continued after 1990, as the second integer flag is set to 1. In this case, IMPORT_TS would provide the data for 1990 for all dates later than 1990.

*This is the same chapter as in the paper

[1]Fortran intrinsic

- The third integer defines the mapping method for time steps in between the points in time defined by the time series data.

  -1: The previous point in time is used.

  0: A linear interpolation between the two nearest points in time is performed.

  1: The next point in time is used.

  In Fig. 3.1 the data for 'exnc' is linearly interpolated, while for `TS(2)` the previous point in time is used.

- The following six integers allow for the selection of a specific date or a specific time span of the data file. The order of entries is `year, month, day, hour, minute, second`. If all six variables are defined one specific date is used independent of the simulation date. If, for example, only the year has been set for a monthly data set, IMPORT_TS cycles over the 12 months of this specific year. Note: the other entries are always deduced from the current date. Thus a simulation using a monthly data set and cycling through one specific year (e.g., 1989 as for `TS(2)`) starting in June would at model start correctly use the data for June. Additionally, it is possible to use, e.g., only 12 UTC data of an hourly data set.

  By default, i.e., all six variables are not set, the data is selected according to the actual simulation date.

- The last float variable defines an offset. The unit of this offset is *days*. With this entry the whole time series can be shifted by a fixed time interval. Thus, for a daily data set defined at 00 UTC, an offset of 0.5 would trigger the usage of new data at 12 UTC instead of 00 UTC.

3.2 Detailed code information

The full information of one data set is contained in a structure of type `T_TS`. This structure consists of two components:

- the information describing the data set read from this `&CTRL_TS` namelist. The information is gathered in a structure of type `T_TS_IO`.
- the data itself.

The type `T_TS_IO` is defined by

```
TYPE T_TS_IO
   !
   ! NAME OF TIME SERIES
   CHARACTER(LEN=STRLEN_OBJECT) :: name = ''
   ! NAME OF FILE WITH DATA
   CHARACTER(LEN=STRLEN_ULONG)  :: fname = ''
   ! VALID RANGE
   REAL(DP), DIMENSION(2)       :: vr = (/ -HUGE(0.0_dp), HUGE(0.0_dp) /)
   ! WHAT TODO IF BEYOND LOWER/UPPER BOUNDARY
   INTEGER, DIMENSION(2)        :: cnt = (/TS_BD_STOP, TS_BD_STOP/)
```

```
    ! INTERPOLATION METHOD
    INTEGER                       :: im = TS_IM_PREV
    !
    ! PICK OUT THIS DATE/TIME (year, month, day, hour, minute, second)
    INTEGER, DIMENSION(6) :: pdt = (/ -1, -1, -1, -1, -1, -1 /)
    !
    ! SHIFT BY THIS NUMBER OF DAYS
    REAL(DP) :: offset = 0.0_dp
    !
  END TYPE T_TS_IO
```

It contains the information listed from the namelist above:

- the name of the special time series (`name`),
- the filename (`fname`),
- the data range (`vr`),
- the "out of time range" policy (`cnt`),
- the interpolation method (`im`),
- the special date (`pdt`) and
- the offset (`offset`).

```
! ------------------------------------------------------------------------------------------
&CTRL_TS
! ### SYNTAX:
!     - name of time series
!     - [var@] name (incl. path) of data file
!           .nc -> netCDF, e.g., "var@my_path_to_my_file/my_file.nc"
!               -> ASCII,  e.g., "my_path_to_my_file/my_file.txt"
!     - valid range ( default: -HUGE(0._dp), HUGE(0._dp) )
!     - out of time interval policy: 0: stop; 1: continue with nearest ...
!       ... (before time interval, after time interval)
!     - interpolation method:  -1: previous; 0: linear interpolation; 1: next
!     - yr,mo,dy,hr,mi,se : pick out always this date/time
!       (example: 2000, , , , , , will cycle through the year 2000 etc.)
!     - offset (in days)
!
! ### EXAMPLE netCDF ###
TS(1) = 'exnc', 'EXNC@/DATA/exnc/EXNC_1950_2012.nc',-99.90,99.90, 0, 0, 0, , , , , , , 0.0,
!
! ### EXAMPLE  ASCII ###
TS(2) = 'exascii','/DATA/example/misc/ex_1985-1990.txt', , , 0, 1, -1, 1989, , , , , , 0.0,
!
/
! ------------------------------------------------------------------------------------------
```

Figure 3.1: Example for the CTRL_TS namelist of IMPORT_TS.

The structure describing the time series data (`T_TS`) comprises in addition to the I/O information the data set information, i.e.

- the number of time steps (`nt`),
- the number of parameters (`np`),
- the time axis in julian days (`jd`),
- the 1D vector containing the information about the parameter axis (`par`), e.g., the level heights,
- the 2D data pointer spanning all parameters and all points in time (`data`),
- a logical flag, if the memory for the output data channel is allocated (lalloc),
- the channel object pointer (`obj`) dimensioned by the length of the parameter axis,
- a flag indicating, if the data is within the (namelist defined) valid range (`flg`), and
- the name of the parameter dimension (`dimname`), i.e., the content of header line 6 (used for an ASCII file) or a 1D-vector of dimension attributes (`dimvaratt`), used in case of netCDF files.

```
TYPE T_TS
   !
   ! IO
   TYPE(T_TS_IO) :: io
   !
   ! NUMBER OF TIME STEPS IN SERIES
   INTEGER :: nt = 0
   !
   ! NUMBER OF PARAMETERS IN SERIES
   INTEGER :: np = 0
   !
   ! TIME AXIS
   REAL(DP), DIMENSION(:), POINTER :: jd => NULL()  ! Julian day + fract.
   !
   ! 'PARAMETER' AXIS
   REAL(DP), DIMENSION(:), POINTER :: par => NULL()
   !
   ! DATA (RANK-1: time, RANK-2: number of parameters)
   REAL(DP), DIMENSION(:,:), POINTER :: data => NULL()
   !
   ! 'CURRENT' VALUE (channel object)
   LOGICAL                           :: lalloc = .FALSE.
   REAL(DP), DIMENSION(:), POINTER :: obj => NULL()
   !
   ! 'FLAG' VALUE (1: OK, 0: OUT OF VALID RANGE)
   REAL(DP), DIMENSION(:), POINTER :: flg => NULL()
   !
   ! NAME OF PARAMETER DIMENSION
   CHARACTER(LEN=STRLEN_MEDIUM) :: dimname  = ''
```

```
    !
    ! ATTRIBUTES OF PARAMETER DIMENSION
    TYPE (t_ncatt), DIMENSION(:), POINTER :: dimvaratt => NULL()
    !
  END TYPE T_TS
```

3.2.1 The SMCL

Apart from the type declarations for `T_TS` and `T_TS_IO`, the SMCL consists of five public and two private subroutines:

- `import_ts_read_nml_ctrl` reads the `&CTRL_TS` namelist (Sect. 3.2.1.1)
- `its_read_ts` analyses the filename string and the data file filling the data structure `TS` with the namelist and file specific values. This subroutine contains two private routines for reading ASCII (`its_read_ts_ascii`) and netCDF (`its_read_ts_netcdf`) files, respectively (Sect. 3.2.1.2).
- `its_copy_io` copys a structure of type `T_TS_IO` to another structure of type `T_TS_IO` (Sect. 3.2.1.3).
- `its_set_value_ts` processes the data (Sect. 3.2.1.4).
- `its_delete_ts` deletes structures of type `T_TS`. More specifically, it releases the memory allocated in `T_TS` (Sect. 3.2.1.5).

The following subsections provide futher details about these subroutines.

3.2.1.1 import_ts_read_nml_ctrl

This subroutine reads the namelist `&CTRL_TS`. The only parameter of the namelist is the variable `TS`, which is an array of structures of type `T_TS_IO` and dimensioned with `NMAXTS = 100`, as a fixed number of entries is required for namelist parameters.

3.2.1.2 its_read_ts

This subroutine reads the data files. First of all the file type (netCDF or ASCII) is determined. For this the filename (`zts%io%fname`) as read in from the `&CTRL_TS` namelist is analysed. If the last three characters in the string equal `'.nc'` a netCDF file is to be read, otherwise an ASCII file. In case of a netCDF file, the string is further broken down to determine the variable name and the filename. If no `@`-sign is found in the string the subroutine returns a non-zero error status. Otherwise, the local filename variable (`fname`) is set to the string behind the `@`-sign, while the local variable name `vname` is set to the first part of the string. In case of an ASCII file, `fname` is set to the full string, while `vname` is an empty string.

After determination of the filename, it is inquired whether the file exists. If not, the subroutine returns a non-zero error status. Otherwise, depending on the extension of the filename, one of the two subroutines `its_read_ts_netcdf` or `is_read_ts_ascii` is called. Upon return from these subroutines, some logfile output is provided. More precise:

- the date range in julian days,

- the range of the data,
- the picked date, and
- the offset

are written. Finally, the memory for the output objects (`zts%obj`) and the output data flag (`zts%flag`) are allocated, if this is requested (i.e., if the third subroutine parameter `lalloc` is `.TRUE.`). This is used in box model applications only, if MESSy CHANNEL is not used and `import_ts_init_memory` is never called.

3.2.1.2.1 SUBROUTINE its_read_ts_netcdf

This subroutine is called if a netCDF file is processed. Parameters of the subroutine are

- a status flag,
- the structure `zts` of type `T_TS`,
- a string containing the filename (`fname`), and
- a string containing the variable name (`vname`) of the respective data set.

Figure 3.2 schematically illustrates the steps required in the subroutine to dimension and import the requested data correctly. More explicitly, the following steps are conducted in the given order:

- The two strings are written to the logfile output telling the name of the variable to be read and the file from which the variable is read.
- The local structure `var` of type `T_NCVAR` is read from the netCDF file using the subroutine `IMPORT_NCVAR`, which is provided by MESSy submodel GRID.
- If the structure does not contain an unlimited dimension, the routine returns with a non-zero error status, as it is required that the variable has a time component.
- If the number of dimensions of the variable is larger than 2, the routine exists with the non-zero error status, because IMORT_TS only deals with 2D data (i.e., time axis x parameter axis).
- If no variable ID is available, the required information is not accessible from the file and thus the subroutine returns with a non-zero error status.
- **Time axis setup:**
  - Determination of the number of time steps: A loop over the dimensions is performed to search for the time dimension (indicated by the UNLIMITED ID in netCDF). If this dimension is found, the length of this dimension is equal to the number of time steps and thus saved in `zts%nt` and written to the logfile.
  - Layout of the time axis: The respective time attributes are imported using the subroutine `IMPORT_NCATT` (hosted by `messy_main_grid_netcdf.f90`). The time unit is first written to the logfile and afterwards parsed by the subroutine `eval_time_str` (hosted by `messy_main_timer.f90`) in order to acquire date and time information from the string[2]

[2] If the string is not successfully parsed, the subroutine returns with an error status.

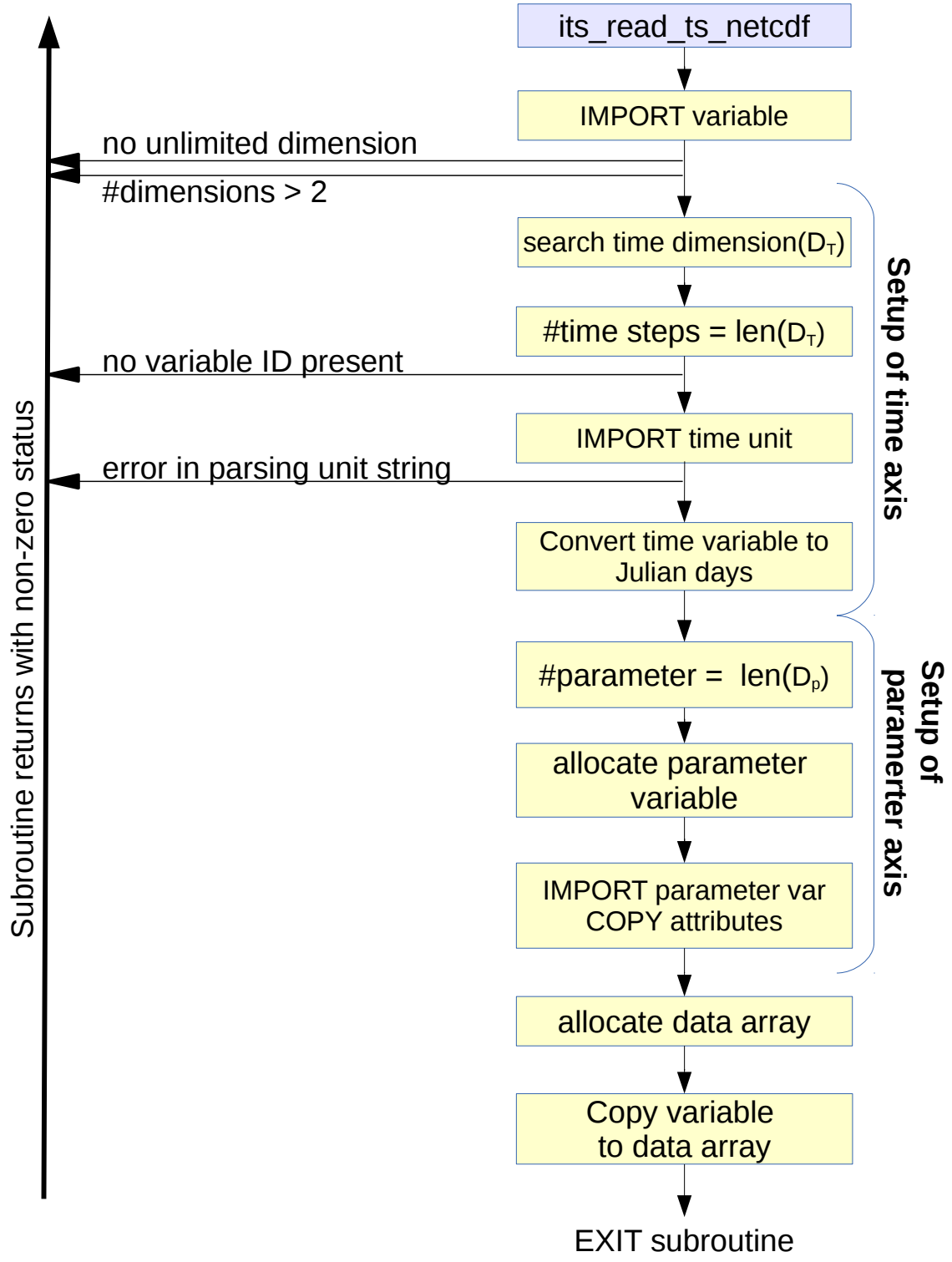

Figure 3.2: Schematic overview of the data processing in the subroutine `its_read_ts_netcdf`.

The time variable is read and converted to the julian day format using the conversion factor and offset determined by the analysis of the netCDF date string and unit[3].

- **Parameter axis setup**
  - The length of the parameter axis (`zts%np`) is given by the length of the second available dimension. The number of parameters is written to the logfile. Knowing the number of parameters the component `zts%par` is allocated to the correct length and preset with a running index.
  - If the parameter axis is present[4], the variable is imported using `IMPORT_NCVAR` and copied to `zts%par`.
  - The name of the dimension is copied to `zts%dimname` and
  - if dimension attributes exist, these are also copied to the respective structure component (`zts%dimvaratt`) using `COPY_NCATT` (available in the module `MESSY_MAIN_GRID_NETCDF`).
  - The content of `zts%par` is dumped to the logfile.
- Finally, the structure component hosting the data (`zts%data`) is allocated and the content of the local structure `var` is copied to the structure describing the data set.

Before leaving the subroutine, the local structure `var` is reinitialised and the status flag is set to zero, indicating success of the subroutine.

3.2.1.2.2 SUBROUTINE its_read_ts_ascii

This subroutine is called, if an ASCII file is processed. Parameters of the subroutine are

- a status flag and
- the structure `zts` of type `T_TS`.

Figure 3.3 schematically illustrates the steps required in the subroutine to dimension and import the requested data correctly. The subroutine analyses the ASCII file in the following order:

- First it is checked, if the structure components `zts%jd`, `zts%par` and `zts%data` are associated. If so, they are deallocated. Afterwards all three pointers are NULLIFYied.
- After acquiring a free unit using the subroutine `FIND_NEXT_FREE_UNIT` from `MESSY_MAIN_TOOLS` the file is opened and the first four header lines are read and ignored.
- From the fifth line the 4 integers time interval flag (`flag`), start year (`y1`), end year (`y2`) and number of parameters (`zts%np`) are read. If `flag` is outside of the range 1 to 6, the subroutine returns with a non-zreo status. Subsequently, the logfile output, providing the time resolution and the number of parameters is produced.

[3]The usual netCDF date format is something like "hour since 2000-08-01 00:00:00". From this the unit ("hours") and the starting date and time ("2000-08-01 00:00:00") are used to calculate the conversion factors for the internal date format julian days. Thus a conversion factor for the time unit (here "hours" to "days") is required (i.e., 24). The offset between the start date and time ("2000-08-01 00:00:00") and the 0th julian day is calculated.

[4]In case of a scalar variable it is not present.

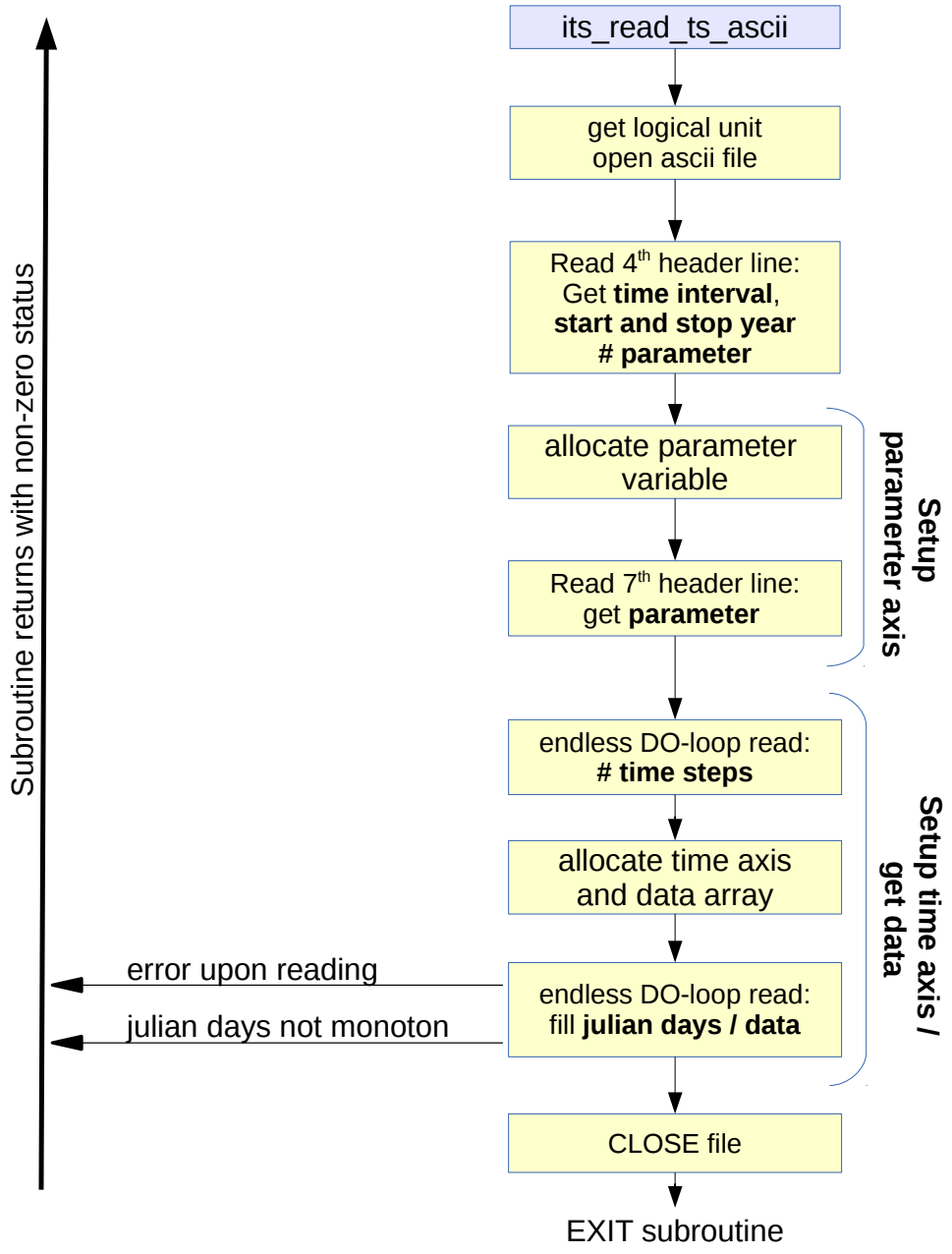

Figure 3.3: Schematic overview of the data processing in the subroutine `its_read_ts_ascii`.

- Parameter axis setup:
  - Knowing the number of parameters, the float vector containing the information about the parameters axis is allocated (`zts%par(zts%np)`).
  - This structure component is filled with the content of line 7 of the data file and the respective information is written to the logfile.
- The header lines 6 and 8 are read, but their content is ignored.

- Determination of the number of time steps: An endless DO-loop is performed, reading one line (i.e. one time step) of the file and enhancing the number of time steps by 1, if the status upon reading is zero. For a negativ status the DO-loop is existed and thus the number of time steps in the file is determined. This information is also written into the logfile. Afterwards, the file is closed.
- Knowing the number of time steps, the structure components containing the julian date (`zts%jd`) and the data (`zts%data`) are allocated.
- Data acquisition: To fill the structure components, the data file is opened again. The 8 header lines are skipped. Subsequently, each data line (date (i)) is processed individually. Depending on the time interval, the respective number of integers (e.g., yearly data = 1 integer; daily data = 4 integers) and the data for this specific point in time are read. If an error occurs upon reading, the subroutine returns with a non-zero status. Using the date components, the julian date (`zts%jd(i)`) is calculated from the newly read date components. If the julian date does not increase monotonically within the file, the subroutine returns with a non-zero status.
- Finally, the data file is closed and the status flag is set to zero.

3.2.1.3 its_copy_io

The subroutine `its_copy_io` copies one structure of type `T_TS_IO` to a second structure of the same type. Therefore it has two parameters of type `T_TS_IO`.

```
! ------------------------------------------------------------------
SUBROUTINE its_copy_io(d, s)

  IMPLICIT NONE

  ! I/O
  TYPE (T_TS_IO), INTENT(OUT) :: d
  TYPE (T_TS_IO), INTENT(IN)  :: s
! ------------------------------------------------------------------
```

The first parameter is the destination structure, the second the source structure. Within the subroutine the components of the structure `T_TS_IO` are copied from the source to the destination. This subroutine is used to fill the `T_TS_IO` structure component of the structure of type `T_TS` with the namelist content.

3.2.1.4 its_set_value_ts

This subroutine performs the data acquisition for the current date.

```
! ------------------------------------------------------------------
SUBROUTINE its_set_value_ts(status, zts, yr, mo, dy, hr, mi, se)

  USE messy_main_timer,        ONLY: gregor2julian

  IMPLICIT NONE
  INTRINSIC :: TRIM
```

```
  ! I/O
  INTEGER,    INTENT(OUT)   :: status
  TYPE(T_TS), INTENT(INOUT) :: zts
  INTEGER,    INTENT(IN)    :: yr, mo, dy, hr, mi, se
! ------------------------------------------------------------------
```

The arguments of this subroutine are:

variable	data type	INTENT	meaning
status	INTEGER	OUT	error flag (zero = success)
zts	TYPE(T_TS)	IN	data structure of specific time series
yr	INTEGER	IN	year of actual simulation time step
mo	INTEGER	IN	month of actual simulation time step
dy	INTEGER	IN	day of actual simulation time step
hr	INTEGER	IN	hour of actual simulation time step
mi	INTEGER	IN	minute of actual simulation time step
se	INTEGER	IN	second of actual simulation time step

The subroutine works as follows:

1. Calculation of the "processing date" / time step to be selected:
   First, the internal data time components are preset with the simulation date and time. Afterwards, the date components "picked" in the namelist are overwritten. This modified date components are converted to julian days, which is the time format the subroutine is internally working with. This date is named the "processing date" in the following.

2. Date check:
   First, it is checked whether the processing date lies within the time range covered by the data file. If not and the "out of time range" policy "0" is requested, the subroutine returns with a non-zero status.

3. Search for the correct time step:
   In a DO-loop over all time steps of the data set, the time step `nt` is searched for which the processing data lies within the interval `zts%jd(nt)` and `zts%jd(nt+1)`.

4. Data processing and check of data range:
   The data handling depends on the chosen interpolation method and on whether the processing date is covered by the data set:

   - If the date is outside the covered time range, but the simulation is continued, the output object `zts%obj` is set to the values of the nearest data point, i.e., the first or the last time step depending on whether the current date is prior or after the date range of the data set. According to the range of valid data regulated in the namelist, a flag (`zts%flg(np)`) is set, indicating whether the data is valid or not. 0 stands for out of range, 1 for valid data.
   - Interpolation method: -1 (previous time step)
     If interpolation method `-1` was chosen in the namelist, the output object `zts%obj` is set to

the content of the data set at time step `nt`. Afterwards each entry `np` of the parameter axis is checked, if the data is in the valid data range and the flag `zts%flg(np)` is set accordingly (0 for out of range and 1 for valid data).

- Interpolation method: 0 (linear interpolation between previous and next time step)
  If linear interpolation between the two nearest points in time is requested, a weighting factor `f` depending on the distance to the two adjacent points in time is calculated and the output data is then weighted between these two points in time according to this factor:

  ```
  f = (jd - zts%jd(nt)) / (zts%jd(nt+1) - zts%jd(nt))
  zts%obj(:) = f * zts%data(nt+1,:) + (1.0 - f) * zts%data(nt,:)
  ```

  In contrast to the two other interpolation methods, it is checked, whether the data of both times of the original data set are within the required range and the flag `zts%flg(np)` is set accordingly.
- Interpolation method: 1 (next time step)
  If interpolation method `1` was chosen in the namelist, the output object `zts%obj` is set to the content of the data set at time step `nt+1`. Afterwards each entry `np` of the parameter axis is checked, if the data is in the valid data range and the flag `zts%flg(np)` is set accordingly (0 for out of range and 1 for valid data).

Finally the status flag is set to 0 indicating the success of the subroutine upon return to the calling routine.

3.2.1.5 its_delete_ts

Within the subroutine `its_delete_ts` the components of a structure of type `T_TS` are deallocated or, depending on the data type, set to their initial values, respectively.

3.2.2 The Basemodel Interface Layer

The BMIL of IMPORT_TS uses four MESSy entry points:

- two in the initial phase,
  - the first for reading and analysing the namelist (`import_ts_initialise`), and
  - the second for memory initialisation (`import_ts_init_memory`),
- one during the integration phase (`import_ts_global_start`), and
- the last one in the finalising phase freeing the memory allocated by the submodel (`import_ts_free_memory`).

3.2.2.1 import_ts_initialise

In the first call during the initialisation phase of MESSy, the namelist `&CTRL_TS` contained in the namelist file `import.nml` is read and analysed. For reading the SMCL subroutine `IMPORT_TS_READ_NML_CTRL` (Sect. 3.2.1.1) is called on the I/O-PE. Afterwards the namelist entries of type `T_TS_IO` are processed on the I/O-PE. First the algorithm loops over the maximum number of input data sets (`NMAXTS`). The algorithm cycles, if the name is empty, as this indicates, that no event with

this number was defined in the namelist. Otherwise the read structure `TS` of type `T_TS_IO` is copied to `XTS(NTS)%io` using the SMCL subroutine `its_copy_io` (Sect. 3.2.1.3), where `XTS` is of type `T_TS`, `io` is its component of type `T_TS_IO`, and `NTS` is a running index for the respective data set. The namelist entries are written to the logfile (see Sect. 3.3.2 for more details). After analysing the namelist entries, the file is opened and analysed using the SMCL subroutine `its_read_ts` (Sect. 3.2.1.2). Within this subroutine the other components of the structure `XTS` are allocated according to the dimensions read from the file.

After all namelist entries have been processed on the I/O-PE, `NTS` is set to the actual number of namelist entries. First, this number is broadcasted to all PEs. In a second loop over all actual data sets, the results of the namelist and file analyses are broadcasted to all PEs and the data arrays are allocated on all PEs.

3.2.2.2 import_ts_init_memory

Here, the memory for the actual output data is allocated in form of channel objects[5].

At the beginning, two list variables `list_dimid` and `list_reprid` are dimensioned according to the number of data sets. In a loop over all data sets, the following steps are required to determine the correct dimensioning of the channel objects for each data set.

1. Determination of the `dimension ID`:
   the correct dimension ID required for the parameter axis needs to be identified, if it already exists, or created, if it does not yet exist. Calling the CHANNEL subroutine `get_dimension_info` it is tested, whether the dimension ID already exists. If so, the subroutine returns with the status flag set to zero and the two variables `DIMID`, providing the respective dimension ID, and `len` providing the length of the dimension. If `len` is not identical to the number of parameters for the respective data set, the dimension name exists already, but not with the correct length. Therefore the dimension name needs to be changed. In this case an arbitrary number between 1 and 1000 is added to the name. In a loop from 1 to 1000 a dimension name is searched, which does not already exist. If an unused dimension name is found, the number is added to the original dimension name

   ```
   XTS(i)%dimname = TRIM(XTS(i)%dimname)//nrstr
   ```

   with `nrstr` the added number as string, and the logical identifier `ldim_ok` is set to `.TRUE.`. `ldim_ok = .TRUE.` indicates, that the dimension needs to be created, while `ldim_ok = .FALSE.` means, that a dimension of the correct length exists. If `ldim_ok = .TRUE.` a new dimension of the length of the parameter axis (`XTS(i)%np`) and the name determined above is defined. The corresponding dimension variable is filled with the float values of `XTS(i)%par` (i.e., the content of line 7 in an ASCII file). Additionally, if dimension attributes are defined in the input file (netCDF file only), they are copied as attributes to the dimension variable. The dimension ID of the newly defined dimension is saved in the variable `list_dimid`. For `ldim_ok = .FALSE.` the dimension ID of the already existing dimension is saved in `list_dimid`.

2. Definition of the `representation`:
   After the dimension is determined, the representation of the channel object needs to be acquired. First it is checked, if the dimension ID of the current data set was already used for a previous data set. In this case, the same representation can be used. If this is not the case, a new

[5] See the Channel User Manual, which is part of the electronic supplement of Jöckel et al., 2010

representation for a 1D variable on an "N"-axis is defined. All respresentation names defined by IMPORT_TS start with the string "TSR_". The representation ID of the newly defined representation is saved in the variable `list_reprid`.

3. Definition of the data channel object:
   Finally, the new channel object and its attributes are defined. The channel is named `import_ts` and the variable name from the namelist is used as channel object name. The information provided by the original namelist entry are defined as namelist attributes for the channel object, i.e., the filename, the valid range, the "out of to time range" policy and the interpolation method.

4. Definiton of the "flag" channel object:
   A second channel object with the same representation is defined, which contains the flag indicating whether the data is in the valid range (`XTS(i)%flag = 1`) or not (`XTS(i)%flag = 0`).

3.2.2.3 import_ts_global_start

This subroutine is called at the beginning of the time loop. Looping over all data sets, the time series data is processed according to the actual simulation date using the SMCL subroutine `its_set_value_ts`.

3.2.2.4 import_ts_free_memory

At the end of the integration the allocated memory (for the variables `list_dimid` and `list_reprid`) is released.

3.3 The stand-alone tool IMPORT_TS

The stand-alone tool of IMPORT_TS is part of the electronical supplement as this User Manual.

3.3.1 Installing stand-alone tool IMPORT_TS

1. unzip the zip-file:
   `unzip import_ts.zip`

2. change into the subdirectory `./import_ts`:
   `cd import_ts`

3. add the necessary entries for your system and compiler to the `Makefile`, i.e.,

`F90`	Fortran90/95 compiler
`F90FLAGS`	Fortran90/95 compiler options (e.g., options for invoking the cpp preprocessor
`DEFOPT`	prefix for compiler directive definitions
`NETCDF_LIB`	netCDF library path
`NETCDF_INCLUDE`	netCDF include path

4. build the executables and modules:
   `gmake`

5. the directory can be cleaned by
   `gmake clean`

The executable `import_ts.exe` should now be available in the main directory. The status after unpacking the zip-file can be archieved with `gmake distclean`.

3.3.2 Running the stand-alone tool IMPORT_TS

After setting up IMPORT_TS as described in the previous section, IMPORT_TS is run by

```
./import_ts.exe import.nml
```

The `import.nml` namelist file should contain at least one entry in the `&CTRL_TS` namelist. An example file is distributed together within the stand-alone tool.

The tool produces the following logfile output:

```
*****************************************************************************
 *** START import: INITIALISATION (import_ts_read_nml_ctrl)
 Reading namelist 'CTRL_TS' from 'import.nml' (unit  101 ) ...
  ... OK !
 *** END   import: INITIALISATION (import_ts_read_nml_ctrl)
 *****************************************************************************
 TIME SERIES   : test
    FILE       : test_data_monthly.txt
    VALID RANGE:  -99.9000000000000057 99.9000000000000057
    BND POLICY : stop
    BND POLICY : stop
    SELECTION  : linear interpolation
    READING DATA ...
     its_read_ts_ascii: OPENING FILE test_data_monthly.txt
     its_read_ts_ascii: TIME RESOLUTION      : month
     its_read_ts_ascii: NUMBER OF PARAMETERES:  19
     its_read_ts_ascii: AXIS                 :  300.000000000000000
        400.000000000000000 500.000000000000000 600.000000000000000
        800.000000000000000 1000.00000000000000 1200.00000000000000
        1500.00000000000000 2000.00000000000000 2500.00000000000000
        3000.00000000000000 3500.00000000000000 4000.00000000000000
        4500.00000000000000 5000.00000000000000 6000.00000000000000
        7000.00000000000000 8000.00000000000000 9000.00000000000000
     its_read_ts_ascii: NUMBER OF TIME STEPS :  642
     its_read_ts_ascii: CLOSING FILE test_data_monthly.txt
     its_read_ts: RANGE OF JULIAN DAY  :  2434743.50000000000  -  2454252.50000000000
     its_read_ts: RANGE OF DATA        :  -41.0000000000000000  -  99.9000000000000057
     its_read_ts: PICK OUT             :  -1 -1 -1 -1 -1 -1
     its_read_ts: OFFSET               :  0.00000000000000000E+00  days
    ... DONE!
 -----------------------------------------------------
```

```
=====================================================
TIME INTEGRATION STARTS ...
 ... DONE!
=====================================================
```

First, the `&CTRL_TS` namelist is read without error. Afterwards the tool loops over all possible `TS` events and produces analytical output. In the example, only one event is found. Its name is 'test' and the file "`test_data_monthly.txt`" is read. The valid data ranges between -90.0 and 90.0. The simulation is stopped, if the simulation date is outside the time span covered by the data file. The data will be linearly interpolated for dates between two data points in the file. Subsequently, the header lines are evaluated: The file contains monthly data. The parameter axis is 19 entries long. The respective values (here heights) are listed in the line labeled AXIS. After the evaluation of the header, the data set is scanned: It is 642 steps long, ranges from julian day 2434743.50 to julian day 2454252.50. The actual range of data is -41.0 to 99.90. No special day is picked and no offset was requested. After the initial phase, the time integration starts. In this case, the model loops from the start date of the data file to the end date and processes the data for all parameters and dates as requested in the namelist. The result is written into an ASCII output file named `output_xx.dat`, where `xx` is a place holder for the index of the processed time series data set.

Bibliography

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.