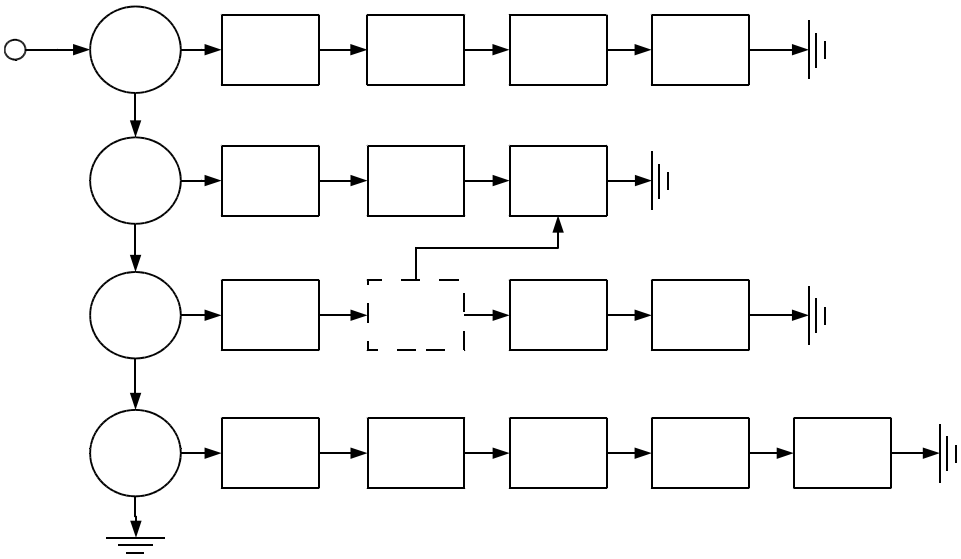
MESSy CHANNEL User Manual

for the MESSy CHANNEL submodel



Patrick Jöckel[1,2], Astrid Kerkweg[3], Andrea Pozzer[4], Rolf Sander[1], Holger Tost[1], Hella Riede[1], Andreas Baumgaertner[1], Sergey Gromov[1], Bastian Kern[1,*]

[1] Air Chemistry Department, Max-Planck Institute of Chemistry, PO Box 3060, 55020 Mainz, Germany

[2] Deutsches Zentrum für Luft- und Raumfahrt, Institut für Physik der Atmosphäre, Oberpfaffenhofen, 822340 Wessling, Germany

[3] Institute for Atmospheric Physics, University Mainz, 55128 Mainz, Germany

[4] Cyprus Institute, EEWRC, P.O. Box 27456, 1645 Nicosia, Cyprus

[*] now at [2]

Patrick.Joeckel@dlr.de

This updated version of the manual is part of the electronic supplement of Kern and Jöckel, "The Modular Earth Submodel System (MESSy, 2.50_extended) as diagnostic interface of the ICOsahedral Non-hydrostatic (ICON) modelling framework", Geosci. Model Dev. (2016). All changes w.r.t. the original manual are marked like this. The original ~~This~~ manual is part of the electronic supplement of our article "Development Cycle 2 of the Modular Earth Submodel System (MESSy2)" in Geosci. Model Dev. (2010), available at: http://www.geoscientific-model-development.net

Date: April 20, 2016

Contents

1 Introduction

This document describes some more details of the MESSy infrastructure submodel CHANNEL for the coupling of processes in Earth System Models. CHANNEL provides the application programming interface (API) of a comfortable and powerful memory management, suitable for the flexible and efficient data exchange / data sharing between different processes (submodels). CHANNEL further serves the input / output of data from / into files, entirely controllable via namelists. The implemented features comprise

- a full input / output control (user interface) via Fortran95 namelists,
- a powerful restart facility for simulation chains,
- output redirection for tailor made output files,
- a flexible choice of the output file format, the output method and the output precision, and
- the capability to conduct basic statistical analyses w.r.t. time on-line, i.e., to output in addition (or alternative) to the instantaneous data the average, standard deviation, minimum, maximum, event counts, and event averages.

For the application of a base model, which contains already the CHANNEL infrastructure, the user interface (namelist control) is explained in Sect. 2.

All other sections provide a reference for the implementation of the CHANNEL infrastructure into new basemodels (in particular Sects. 4, 5.5.1, 6 and 7.4), and for the application of the CHANNEL infrastructure in submodels, when the basemodel already contains the CHANNEL infrastructure.

CHANNEL is written in Fortran95 (ISO/IEC-1539-1) following an object oriented approach to the extent possible. The basic entities (implemented as Fortran95 structures) in CHANNEL are

- *attributes*, → time independent, scalar characteristics,
- *dimension variables*, → specific coordinate axes,
- *dimensions*, → the basic geometry in one dimension
- *representations*, → multidimensional geometric structures (based on *dimensions*),
- *channel objects*, → data fields including their meta information (*attributes*) and their underlying geometric structure (*representation*)
- *channels*, → sets of "related" *channel objects* with additional meta information. The "relation" can be, for instance, the simple fact that the *channel objects* are defined by the same submodel.

These structures are explained in more detail in Sect. 3 for reference. Direct access to the structures is not required, since CHANNEL provides a powerful application programming interface (API), as explained in Sects. 4 and 5. The specific coupling of the MESSy infrastructure submodel TRACER to CHANNEL is explained in Sect. 6.

For data input / output, currently interfaces for netCDF[1], ~~and~~ parallel netCDF[2], and CDI[3] are implemented, the implementation of alternative file formats is straightforward due to the modular structure of CHANNEL. This is further explained in Sect. 7.

2 CHANNEL namelist user interface

According to the MESSy standard (Jöckel et al., 2005)[4], the user interface of the submodel CHANNEL is subdivided into a control (`CTRL`) and a coupling (`CPL`) namelist, where the latter contains base model specific control parameters and a coupling to the MESSy infrastructure submodel TIMER for the time control of output files. Both namelists are contained in the namelist file *channel.nml*.

[1] `http://www.unidata.ucar.edu/software/netcdf`
[2] `http://www.mcs.anl.gov/parallel-netcdf`
[3] `https://code.zmaw.de/projects/cdi`
[4] `http://www.atmos-chem-phys.net/8/1677`

2.1 CHANNEL CTRL namelist

The CTRL namelist comprises entries to control the output and restart handling for all *channels* and *channel objects*. An example is given in Figure 1.

```
&CTRL
!
EXP_NAME='test'       ! EXPERIMENT NAME
!
L_FLUSH_IOBUFFER = F, ! FLUSH I/O BUFFER IN EVERY TIME STEP
                      ! (DEFAULT: T (true))
!
OUT_PREC  = 1, 1, 1, 1, 1, 1, 1 ! OUTPUT PRECISION (production)
!OUT_PREC = 1, 2, 2, 1, 1, 1, 2 ! OUTPUT PRECISION (for tests)
!
! # SET DEFAULT OUTPUT AND RESTART HANDLING
!      - '', OUTPUT-FILETYPE, RERUN-FILETYPE,
!        NO. OF STEPS PER OUTPUT-FILE, RERUN, IGNORE,
!        INST, AVE, STD, MIN, MAX, CNT, CAV, RANGE(2)
!
OUT_DEFAULT = '', 3, 3, -1, F,F, T,F,F,F,F, F,F, , ,
!
! # ADD NEW OUTPUT CHANNELS
ADD_CHANNEL(1)   = 'special',
ADD_CHANNEL(2)   = 'carbon',
!
! # ADD NEW CHANNEL OBJECT REFERENCES
!   NOTES:
!    - SOURCE OBJECT NAME MAY CONTAIN WILDCARDS '?' AND '*'
!      (in this case, target object name is ignored)
!    - TARGET CHANNEL NAME MAY CONTAIN WILDCARDS '?' AND '*'
!    - TARGET OBJECT NAME SET TO SOURCE OBJECT NAME, IF ''
!
ADD_REF(1) = 'g3b',      'aps',    '*',         '',
ADD_REF(2) = 'g3b',      'geopot', 'tracer_gp', 'geop',
ADD_REF(3) = 'tracer_gp','C*',     'carbon',    '',
!
! # SET CHANNEL SPECIFIC DEFAULT OUTPUT AND RESTART HANDLING
!      - channel-name, OUTPUT-FILETYPE, RERUN-FILETYPE,
!        NO. OF STEPS PER OUTPUT-FILE, RERUN, IGNORE,
!        INST, AVE, STD, MIN, MAX, CNT, CAV, RANGE(2)
!   NOTE: IF NO. OF STEPS PER OUTPUT-FILE <= 0,
!         THE EVENT TRIGGER (CPL-NAMELIST, TIMER_TNF BELOW)
!         CONTROLS THE FILE CONTENT
!
OUT_CHANNEL(1) = 'g3b',    3, 3, 10, F,F, T,F,F,F,F, F,F, , ,
OUT_CHANNEL(2) = 'qtimer', 3, 3, -1, F,F, T,T,T,T,T, F,F, , ,
OUT_CHANNEL(3) = 'special',3, 3, -1, F,F, F,T,F,F,F, F,F, , ,
!
! # SET CHANNEL OBJECT SPECIFIC OUTPUT AND RESTART HANDLING
!      - channel-name, object-name, RERUN, IGNORE,
!        INST, AVE, STD, MIN, MAX, CNT, CAV, RANGE(2)
!
OUT_OBJECT(1)  = 'g3b',   'aps',     F,F, T,F,F,F,F, F,F, , ,
OUT_OBJECT(2)  = 'carbon','C2H4',    F,T, T,T,F,F,F, F,F, , ,
/
```

Figure 1: Example CTRL namelist of the generic submodel CHANNEL. For explanations, see text.

Four entries control the overall output:

- EXP_NAME is an arbitrary string (of maximum 15 characters) which defines the simulation. This string (padded with trailing underscores) represents the first part of the output filenames, followed by an underscore. The output filename is completed by the simulation date and time (format YYYYMMDD_hhmm) of the first entry in the file, another underscore, the name of the *channel*, and the file format specific extension (including the dot), in summary *EXP_NAME*_YYYYMMDD_hhmm_*channel.ext*.
  Example: EXP_NAME = 'test',

results for instance in the netCDF (.nc) filename test___________20090101_0030_tracer_gp.nc for the *channel* tracer_gp. The first output time step in this file corresponds to the simulation time January 1, 2009 at 00:30 UTC. Additional digits for seconds (2) and for milliseconds (3) are added to the output file name for models with temporal finer resolution. The format for these models is *EXP_NAME*_YYYYMMDD_hhmmssss_*channel.ext*.

- L_FLUSH_IOBUFFER is a logical switch, which controls the flushing of the output memory buffers. If set to T (default) the output buffers of all output files are flushed at the end of each output time step, otherwise (F) the flushing is determined by the runtime environment (i.e., if the buffer is full). For instance, if netCDF is used as output format, flushing every output time step might reduce the overall runtime performance, however, it guarantees valid files even after a crash of the simulation.
  Example: L_FLUSH_IOBUFFER = T,

- OUT_PREC is a one dimensional array of length 7 ~~6~~ for setting the output precision for the output file formats (ASCII, parallel netCDF, netCDF, GRIB, HDF4, HDF5, CDI), respectively. Currently only netCDF, ~~and~~ parallel netCDF, and netCDF output via CDI (7) are implemented. For netCDF and parallel netCDF, **1** sets the output to NF90_FLOAT and **2** to NF90_DOUBLE, respectively. Output in double precision is helpful for performing restart-tests, i.e., for checking that the simulation results are independent of the chosen restart frequency in simulation chains.
  Example: OUT_PREC = 1,2,1,1,1,1,2,

- OUT_DEFAULT is a structure controlling the default output settings of all *channels* and *channel objects*; the elements are

  1. an empty string (this string is used for *channel* specific settings (see OUT_CHANNEL below),
  2. an integer to control the output file-type (currently only parallel netCDF (2), ~~and~~ netCDF (3), and netCDF output via CDI (7) are implemented),
  3. an integer to control the restart file-type (currently only parallel netCDF (2) and netCDF (3) are implemented),
  4. an integer to control the maximum number of time-steps per output file (if this number is reached, a new output-file is started),
  5. a logical to force output into the restart file,
  6. a logical to ignore, if an object is not present in a restart file, although required (this is useful if additional submodels are switched on after a restart),
  7. a logical to output instantaneous data (i.e., the values at the output time step, INST),
  8. a logical to output the average w.r.t. time of the data between two subsequent output time steps (AVE),
  9. a logical to output the standard deviation w.r.t. time of the data between two subsequent output time steps (STD),
  10. a logical to output the minimum w.r.t. time of the data between two subsequent output time steps (MIN),
  11. a logical to output the maximum w.r.t. time of the data between two subsequent output time steps (MAX),
  12. a logical to count (and output) the number of events defined by the data being within a specific range between two subsequent output time steps (CNT),
  13. a logical to average over the events defined by the data being within a specific range between two subsequent output time steps (CAV),
  14. a pair of real numbers describing the interval boundaries for CNT and CAV, the defaults are -HUGE(0.0_DP) and HUGE(0.0_DP), respectively.

  The corresponding variable names in the output (and restart) files are the *channel object* names extended by an underscore and the suffixes (in lowercase) listed in parentheses, except for INST, for which no extension is used.
  Example: OUT_DEFAULT = '', 2, 2, 10, F,F, T,F,F,F,F, F,F, , ,
  specifies parallel netCDF output as file-type for output (2) and restart files (2), respectively; a maximum of 10 output time steps are written to each output file. The objects without lrestreq=.TRUE. (see Sects. 3.5, 5.5.2.1 and 5.5.3.1) set in the code are not written to the restart files (F). After a restart *channel objects* with lrestreq=.TRUE. which are not present in the respective restart file will not be ignored (F), but rather cause an error. The instantatneous data of the *channel objects* are output (T), but none of the derived statistics (F,F,F,F, F,F). The standard range for the conditional statisitcs (CNT and CAV) remains default (, ,).

For models with more than one nested patch and internal coupling, the *channel names* in each patch are identical. During output a unique filename is created from the *channel name* and the number of the patch.

The settings in `OUT_DEFAULT` can be overwritten for specific *channels* and / or specific *channel objects*, respectively:

- `OUT_CHANNEL(n)` is used for *channels*, where n is an arbitrary but unique number between 1 and 500. The same entries (1-14) as in `OUT_DEFAULT` are used, but the empty string (entry 1) is replaced by the name of the *channel*.
  Example: `OUT_CHANNEL(1) = 'geoloc', 2, 2, 10, F,F, F,F,F,F,F, F,F, , ,`
- `OUT_OBJECT(n)` is used for *channel objects*, where n is an arbitrary but unique number between 1 and 1000. The entries correspond to the entries 5-14 of `OUT_DEFAULT`, preceded by two strings denoting the name of the *channel* and the name of the *channel object*, respectively.
  Example: `OUT_OBJECT(1) = 'g3b','qtnew', F,F, F,F,F,F,F, F,F, , ,`

In a standard setup, all *channel objects* in a specific *channel*, which are subject to output (i.e., for which not all of the output flags above (entries 7-13) are `F`), are written to the same output file. The possibility exists, however, to re-direct the output of *channel objects* into different output files by creating *channel object references* in the namelist with the variable

- `ADD_REF(n)`, where n denotes an arbitrary but unique number between 1 and 500 and each *reference* consists of four strings denoting
  - the name of the original *channel* which contains the *channel object* to be referenced,
  - the name of the *channel object* to be referenced,
  - the name of the target *channel*, i.e., the *channel* the object should be referenced from,
  - the name of the *channel object* in the referencing *channel*. If this entry is empty, the original name of the *channel object* is used.

  Example: `ADD_REF(1) = 'g3b', 'aps', 'tracer_gp', 'ps',`
  creates in *channel* `tracer_gp` a reference (with name `ps`, to the *channel object* `aps` of *channel* `g3b`.

A special feature is that the name of the target *channel* can contain wildcards ('*' or '?'). The *channel object reference* is then created in every *channel* with matching name. For example
`ADD_REF(1) = 'g3b', 'aps', '*', '',`
creates in every *channel* (`*`) a reference (with name `aps`, since the last string is empty, `''`) to the *channel object* `aps` of *channel* `g3b`. References defined with `ADD_REF` are valid for every nested patch.

Also the name of the *channel object* to be referenced can contain wildcards ('*' or '?'). In this case, all matching *channel objects* from the original *channel* will be referenced from the target *channel* – with the name of the original *channel object*. This implies that in this case, a potentially specified *channel object* name in the referencing *channel* (4th entry) is ignored. For example
`ADD_REF(2) = 'tracer_gp', 'C*', 'carbon', '',`
creates in the `carbon` *channel* references for all *channel objects* of the `tracer_gp` *channel* which names start with 'C'.

The *channel object reference*(s) share(s) with the original *channel object* the memory for the primary data, i.e., no additional memory is consumed. Memory for secondary (derived statistical) data, however, is created for each *channel object reference* as it is requested by the corresponding namelist entries (`OUT_DEFAULT`, `OUT_CHANNEL(n)`, `OUT_OBJECT(n)`). This additional memory is required to allow independent output time intervals and statistics for the *channel object reference*(s).

In addition to the creation of *channel object references* for output redirection into other already existing *channels*, also new channels can be created with the namelist variable

- `ADD_CHANNEL(n)`, where n denotes an arbitrary but unique number between 1 and 50, and each new *channel* is specified by its unique name.
  Example: `ADD_CHANNEL(1) = 'special',`

As indicated above (entry 4 of `OUT_DEFAULT` and / or `OUT_CHANNEL(n)`), the size of the output files is controlled via the maximum number of time steps per file. If this number is set to zero or negative, the files are controlled via the entries in the `CPL` namelist as described in the next section.

```
&CPL
!
L_BM_ORIG_OUTPUT = F, ! ENABLE ORIGINAL LEGACY MODEL OUTPUT ?
!
! # SET OUTPUT INTERVALS FOR ALL CHANNELS (DEFAULT + INDIVIDUAL)
!   NOTE: First match (wildcard) counts!
!
TIMER_DEFAULT    = '',     5, 'hours',   'first', 0,
!
TIMER_CHANNEL( 1) = 'qtimer',  1, 'steps',   'first', 0,
TIMER_CHANNEL( 2) = 'scout',   1, 'hours',   'first', 0,
TIMER_CHANNEL( 3) = 'special', 1, 'months',  'first', 0,
!
! # SET TIMER EVENTS FOR NEW FILENAMES
!   (IF NO. OF STEPS PER OUTPUT-FILE <= 0 ABOVE !!!)
!   NOTE: First match (wildcard) counts!
!
TIMER_TNF_DEFAULT = '', 1, 'days', 'first', 0,
!
TIMER_TNF_CHANNEL( 1) = 'qtimer', 1, 'days',   'first', 0
TIMER_TNF_CHANNEL( 2) = 'scout*', 1, 'months', 'first', 0
!
/
```

Figure 2: Example CPL namelist of the generic submodel CHANNEL. For explanations, see text.

2.2 CHANNEL CPL namelist

An example for the `CPL` namelist of CHANNEL is given in Figure 2.

The `CPL` namelist of CHANNEL provides the switch `L_BM_ORIG_OUTPUT` (default `F`) to enable (`T`) the original output of the basemodel, in case the base model supports this.

The more important features, however, are provided by a coupling to the MESSy infrastructure submodel TIMER. This enables the control of the output frequency and the file sequence containing a time series.

The default output frequency for all *channels* is set by the variable `TIMER_DEFAULT`, which can be overwritten for each *channel* by the variable `TIMER_CHANNEL(n)`, where n indicates an arbitrary but unique number between 1 and 500. The frequency to create new files is set by the variable `TIMER_TNF_DEFAULT` (default) for all *channels*, and overwritten for specific *channels* by the variable `TIMER_TNF_CHANNEL(n)`, respectively. Also here n denotes an arbitrary but unique number between 1 and 500.

All four variables consist of

- the name of the *channel*, which is empty for the DEFAULT variables,
- an event trigger, which consists of
  - the length of the time interval (integer),
  - the unit of the time interval (string), e.g., 'steps', 'minutes', 'hours', 'days', 'months', 'years',
  - the positioning of the triggering time step in the given interval, i.e., 'first', 'last', or 'exact',
  - an offset in seconds (integer), which is usually zero.

As an example, the namelist entry

```
TIMER_CHANNEL( 1) = 'g3b', 5, 'hours', 'first', 0
```

triggers the output of all *channel objects* in the *channel* 'g3b', for which output is requested, at the 'first' time step in every 5 hours with no offset. The entry

```
TIMER_TNF_DEFAULT = '', 1, 'days', 'first', 0,
```

triggers a new file creation per default for all *channels* at the 'first' time step of every day without offset.

This feature for controlling new file creation is only active, however, for those channels, for which the number of time steps per output file in the `CTRL` namelist (entry 4 of `OUT_DEFAULT` and / or `OUT_CHANNEL(n)`) is set to zero or negative.

2.3 CHANNEL CTRL_PNETCDF namelist

The CTRL_PNETCDF namelist comprises entries to tune the performance of the parallel netCDF output via MPI-IO hints in the form MPI_IO_HINT(n) = hint,value where n denotes an arbitrary but unique number between 1 and 10, *hint* is the name of the MPI-IO hint, and *value* its value. Example:

```
&CTRL_PNETCDF
  MPI_IO_HINT(1) = 'IBM_sparse_access','true',
/
```

3 Type definitions of basic entities

The basic entities used within the MESSy CHANNEL infrastructure submodel are

- *attributes*,
- *dimension variables*,
- *dimensions*,
- *representations*,
- *channel objects*,
- *channels*.

These entities are hierarchically organised in a way that a higher entity (later in list) can make use of a lower entity (earlier in list). This section provides a complete reference of the corresponding Fortran95 structures. It is important to note that the application of the MESSy submodel CHANNEL does not require any direct access to these structures in the code. Application of the submodel CHANNEL is possible by exclusive usage of the interface subroutines which are explained in the subsequent sections.

Figure 3 sketches the relationship between the different entities and the corresponding Fortran95 modules.

Fortran parameters defined in messy_main_constants_mem.f90 used for the type declarations are:

```
! PRECISION SETTINGS
INTEGER, PARAMETER :: DP = SELECTED_REAL_KIND(12,307)
INTEGER, PARAMETER :: I8 = SELECTED_INT_KIND(14)

! STRING LENGTHs
INTEGER, PARAMETER :: STRLEN_SHORT  = 8
INTEGER, PARAMETER :: STRLEN_MEDIUM = 24
INTEGER, PARAMETER :: STRLEN_LONG   = 64
INTEGER, PARAMETER :: STRLEN_VLONG  = 80
INTEGER, PARAMETER :: STRLEN_ULONG  = 256
```

3.1 Attributes

An *attribute* describes a time independent, scalar characteristic of a higher entity. The information is stored in a Fortran95 structure:

```
TYPE t_attribute
   CHARACTER(LEN=STRLEN_MEDIUM) :: name    = ''
   INTEGER                      :: type    = TYPE_UNKNOWN
   INTEGER                      :: iflag   = AF_NONE
   INTEGER                      :: i = 0
   CHARACTER(LEN=STRLEN_ULONG)  :: c = ''
   REAL(DP)                     :: r = 0.0_DP
END TYPE t_attribute
```

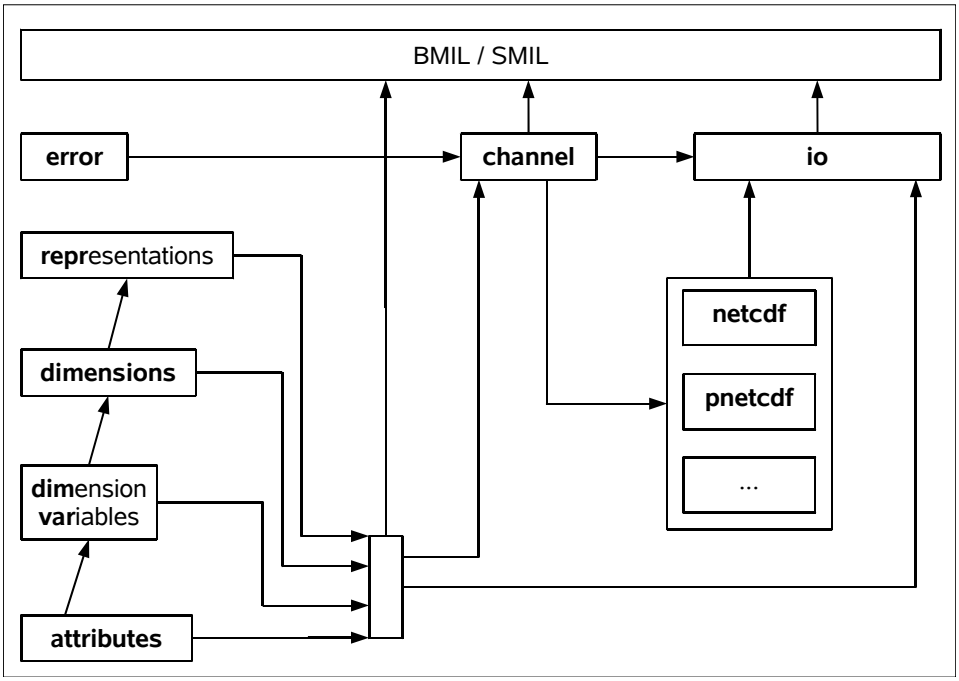


Figure 3: Relationship between the various channel entities and Fortran95 modules: The boxes represent the different Fortran95 modules of CHANNEL, the corresponding filenames are `messy_main_channel_`*bf*`.f90`, where *bf* is the respective bold-face text. The arrows indicate where the different modules are `USE`d. Different output formats / methods (netcdf, pnetcdf (for parallel-netCDf), and other formats / methods indicated by the dots) are summarised in a common box for the sake of readability to reduce the number of arrows. Similarly, the empty box summarises similar uses of the connected modules. BMIL and SMIL denote the basemodel layer and the submodel interface layer, respectively.

Attributes are identified by their `name` and are of a unique `type` (pre-defined with the Fortran95 parameter `TYPE_UNKNOWN`), which is either `INTEGER` (parameter `TYPE_INTEGER`), `REAL(DP)` (parameter `TYPE_REAL_DP`) or `CHARACTER(LEN=STRLEN_ULONG)` (parameter `TYPE_STRING`). A set of attributes of a higher entity is stored in a concatenated list of attributes:

```
TYPE t_attribute_list
   TYPE(t_attribute)               :: this
   TYPE(t_attribute_list), POINTER :: next => NULL()
END TYPE t_attribute_list
```

Basic subroutines for managing *attributes* and *attribute lists* are contained in the Fortran95 module `messy_main_channel_attributes.f90` (Sect. 5.1). The Fortran95 module `messy_main_channel.f90` (Sect. 5.5) contains additional subroutines for handling *attribute lists* of higher entities.

3.2 Dimension variables

A *dimension variable* contains the information of a discrete coordinate, such as an axis of a spatial dimension (e.g., latitude). The required information is stored in a Fortran95 structure:

```
TYPE t_dimvar
   CHARACTER(LEN=STRLEN_MEDIUM)    :: name  = ''    ! NAME OF VARIABLE
   REAL(DP), DIMENSION(:), POINTER :: val => NULL() ! VALUES
   TYPE(t_attribute_list), POINTER :: att => NULL() ! ATTRIBUTE LIST
END TYPE t_dimvar
```

Each *dimension variable* is identified by its `name`. The coordinate values are stored in the array `val`. The *attribute list* contains further information, such as for instance the unit of the coordinate values (e.g., "unit"="degrees_north" for the latitude).

Dimension variables are necessarily associated with *dimensions* (Sect. 3.3). Since more than one *dimension variable* can be associated with a specific *dimension*, lists of *dimension variables* are stored in a concatenated Fortran95 structure:

```
TYPE t_dimvar_list
   TYPE(t_dimvar)               :: this
   TYPE(t_dimvar_list), POINTER :: next => NULL()
END TYPE t_dimvar_list
```

Subroutines for managing *dimension variables* and *dimension variable lists* are contained in the Fortran95 module `messy_main_channel_dimvar.f90` (Sect. 5.2).

3.3 Dimensions

The term *dimension* is self-explanatory. The structure is:

```
  TYPE t_dimension
   CHARACTER(LEN=STRLEN_MEDIUM) :: name  = ''          ! NAME OF DIMENSION
   INTEGER                      :: id    = DIMID_UNDEF ! ID OF DIMENSION
   INTEGER                      :: len   = 0           ! LENGTH OF DIMENSION
   LOGICAL                      :: ltime = .FALSE.     ! FLAG FOR TIME DIM.
   TYPE(t_dimvar_list), POINTER :: var   => NULL()     ! DIMENSION VARIABLES
END TYPE t_dimension
```

A *dimension* is identified by its unique `name` and identifier (`id`). The basic information hold by a *dimension* is its length (`len`), which is the number of discrete steps along the axis. A special flag is used to indicate that the *dimension* is the time dimension (`ltime`). Specific coordinate values may be stored in a list of *dimension variables* (`var`, Sect. 3.2).

All *dimensions* are stored internally in one list. Subroutines for managing *dimensions* are contained in the Fortran95 module `messy_main_channel_dimensions.f90` (Sect. 5.3).

3.4 Representations

A *representation* describes the underlying geometric structure of data. The required information is stored in a Fortran95 structure:

```
TYPE t_representation
   ! IDENTIFICATION
   CHARACTER(LEN=STRLEN_MEDIUM) :: name     = ''   ! NAME
   INTEGER                      :: id       = 0    ! ID
   INTEGER                      :: patch_id = -1   ! patch
   INTEGER                      :: rank     = 0    ! RANK
   CHARACTER(LEN=IRANK)         :: link     = ''   ! LINK-STRING
   CHARACTER(LEN=IRANK)         :: axis     = ''   ! AXIS-STRING
   INTEGER, DIMENSION(IRANK)    :: gdimlen  = 0    ! GLOBAL DIMENSION LENGTH
   !
   ! DECOMPOSITION INFORMATION
   INTEGER, DIMENSION(IRANK)    :: ldimlen  = 0    ! LOCAL DIMENSION LENGHT (ON THIS PE)
   INTEGER                      :: dctype   = 0    ! DECOMPOSTION TYPE
   !
   ! FULL DIMENSION INFORMATION (POINTER TO DIMENSION; GLOBAL !!!)
   TYPE(t_dimension_ptr), DIMENSION(IRANK) :: dim
   !
   ! INPUT/OUTPUT CONVERSION
```

```
    ! - PERMUTATION OF DIMENSIONS BEFORE OUTPUT
    LOGICAL                    :: lperm          = .FALSE.
    INTEGER, DIMENSION(IRANK) :: order_mem2out  = 0 ! output
    INTEGER, DIMENSION(IRANK) :: shape_out      = 0 ! shape in output
    INTEGER, DIMENSION(IRANK) :: order_out2mem  = 0 ! input
    !
    ! BOUNDARY MEMORY MANAGEMENT
    TYPE(t_repr_boundary)     :: bounds
    !
    ! PARALLEL DECOMPOSITION INFORMATION
    TYPE(t_repr_pdecomp)      :: pdecomp
    !
    INTEGER :: hgrid                = -1
    INTEGER :: vgrid                = -1
    !
  END TYPE t_representation
```

Each *representation* is identified by a unique `name` and identifier (`id`), `patch_id` is additionally stored for models with nested patches and internal coupling, which allows to use identical names in different *patches*. In this case a particular *representation* can be identified by its `id` or the combination of its `name` and the `patch_id`. The `rank` of the data determines the number of *dimensions* (`dim`) that span the *representation*. All *channel object* data are internally stored in arrays of rank `IRANK` (=4). This implies that representations can be of rank 0 (scalar) to 4. The internal ranks used by the data (active *dimensions*) are stored in `link`. Active ranks are marked by 'x', inactive ranks by '-'. For example, 'xx--', 'x-x-', 'x--x', '-xx-', etc. are valid `link`s for data of rank 2. The corresponding dimension lengths are stored in `gdimlen`.

Specific geometric axes are identified with `axis`, 'X', 'Y' and 'Z' denote the spatial axes, 'N' an arbitrary index axis, and '-' must be used, if the axis has no specific meaning. For instance 'XYZ-' will identify that the ranks 1 to 3 of the representation correspond to the Eulerian spatial dimensions. This information can be used for generic transformation and transposition routines.

In parallel environments, data can be distributed to several processes. The 'local' (i.e., parallel decomposed) dimension length is stored in `ldimlen`. The flag `dctype` is used to identify the corresponding subroutines for collecting data from all parallel processes (gather) and for distributing data to all parallel processes (scatter), respectively.

Depending on the output format and the memory layout, it might be desirable to reshape data arrays for output. The vectors `shape_mem` and `shape_out` describe the array dimensions in memory and output, respectively. If it is required to permute ranks between memory and output, this is indicated by `lperm`. The corresponding permutations are stored in `order_mem2out` for output, and `order_out2mem` for input, respectively.

Another feature, which is useful for parallel decompostions, is the possibility to increase the 'local' (i.e., parallel decomposed) dimension length by additional boundary points, sometimes called 'ghost' points. These are for instance applied, if gradients along the specific dimension are required for the calculations over the parallel processes. The data on these boundaries are then regularly exchanged by the 'neighbouring' parallel processes. Information about such additional local boundaries are stored in the structure component `bounds`, which itself is a Fortran95 structure:

```
  TYPE t_repr_boundary
     ! HOLDS INFORMATION ABOUT BOUNDARIES
     LOGICAL                    :: lbounds = .FALSE.
     !
     ! number of boundary indices
     INTEGER, DIMENSION(IRANK) :: nbounds = 0
     !
  END TYPE t_repr_boundary
```

`lbounds` is `.TRUE.`, if the corresponding *representation* has been defined with additional local boundaries in at least one *dimension*. Without additional local boundaries `lbounds` is `.FALSE.`. `nbounds` contains the number of additional points (always symmetrically added at both ends) for each *dimension*.

If the system is capable for parallel input / output (e.g., parallel netCDF, Sect. 7.3) additional decomposition information is stored in `pdecomp`, a Fortran95 structure that maps contiguous sub-arrays (segments) to the global output array:

```
TYPE t_repr_pdecomp
   ! HOLDS INFORMATION ABOUT PRALLEL DECOMPOSITION
   LOGICAL                  :: lpdecomp = .FALSE.  ! info available ?
   !
   ! (UN-DEFORMED (e.g. DE-VECTORISED)) SHAPE IN MEMORY
   INTEGER,DIMENSION(IRANK) :: shape_mem = 0 ! shape in memory (local)
   INTEGER,DIMENSION(IRANK) :: shape_out = 0 ! shape for output (local)
   !
   ! MAPPING BETWEEN GLOBAL AND LOCAL
   INTEGER                           :: nseg = 0        ! number of segments
   INTEGER, DIMENSION(:,:), POINTER :: ml    => NULL() ! lower bounds in mem
   INTEGER, DIMENSION(:,:), POINTER :: mu    => NULL() ! upper bounds in mem
   INTEGER, DIMENSION(:,:), POINTER :: start => NULL() ! start in output
   INTEGER, DIMENSION(:,:), POINTER :: cnt   => NULL() ! count in output
   !
   ! SPECIAL FOR PERFORMANCE TUNING ...
   INTEGER :: piotype = PIOTYPE_IND
   !
END TYPE t_repr_pdecomp
```

If the additional decomposition information is not provided for a specific *representation* with the subroutine `set_representation_decomp` (see Sect. 5.4.2), `lpdecomp` remains `.FALSE.` and parallel input / output is not possible. The array shapes in memory and for the output are stored in `shape_mem` and `shape_out`, respectively. The number of contiguous segments (in the output array) for the specific parallel process is `nseg`. Each segment is described by its bounds in memory (`ml` and `mu`) and its `start` and count (`cnt`) index vectors in the output array. The parallel output type (`piotype`) can be `PIOTYPE_SGL` for single process input / output, `PIOTYPE_IND` for independent input / output, and `PIOTYPE_COL` for collective input / output.

For CDI, the identifiers of the horizontal and the vertical grid are stored in `hgrid` and `vgrid`, respectively.

All *representations* are stored centrally in one list. Subroutines for managing *representations* are contained in the Fortran95 module `messy_main_channel_repr.f90` (Sect. 5.4).

3.5 Channel objects

A *channel object* describes data and its meta information defined in a Fortran95 structure:

```
TYPE t_channel_object
   CHARACTER(LEN=STRLEN_OBJECT)     :: name  = ''     ! NAME
   TYPE(t_attribute_list), POINTER  :: att => NULL()  ! OBJECT ATTRIBUTES
   TYPE(t_representation), POINTER  :: repr => NULL() ! REPRESENTATION
   TYPE(t_channel_object_mem)       :: memory         ! MEMORY MANAGEMENT
   TYPE(t_channel_object_io)        :: io             ! I/O MANAGEMENT
   ! ABSOLUTELY REQUIRED IN RESTART FILE ?
   LOGICAL                          :: lrestreq = .FALSE.
   TYPE(t_channel_object_int)       :: int            ! FOR INTERNAL USE
   REAL(DP), DIMENSION(:,:,:,:),     POINTER :: data => NULL() ! DATA
   TYPE(PTR_4D_ARRAY), DIMENSION(:), POINTER :: sdat => NULL() ! 2ndary DATA
   ! POINTER TO REGION WITHOUT BOUNDARIES FOR I/O
   REAL(DP), DIMENSION(:,:,:,:),     POINTER :: ioptr => NULL()
END TYPE t_channel_object
```

A *channel object* is identified by a unique `name` (the string length `STRLEN_OBJECT` is `2*STRLEN_MEDIUM + 5`) in this specific *channel*, additional properties are described by a *list of attributes* (`att`, Sect. 3.1) and the underlying geometric structure is described by a pointer to the corresponding *representation* (`repr`, Sect. 3.4). Whether a *channel object* is required to be dumped into a restart file for chain simulations or not, is controlled by the switch `lrestreq`. The pointer `data` points to the primary data memory, the pointer array `sdat` to the secondary (derived statistical) data

memory, respectively. The pointer `ioptr` (internally used for I/O) points to the internal sector (i.e., the part inside the additional boundaries) of the primary data memory, in case the underlying *representation* is specified with additional boundaries. If no boundaries are present, `ioptr` is identical with `data`.

The Fortran95 structure `memory`

```
TYPE t_channel_object_mem
   !
   ! MEMORY USAGE
   INTEGER(I8) :: usage       = 0_I8  ! primary data section
   INTEGER(I8) :: usage_2nd   = 0_I8  ! secondary data section
   !
   ! FLAGS FOR INTERNAL USE
   LOGICAL     :: lalloc  = .FALSE.   ! AUTOMATIC MEMORY ALLOCATION
END TYPE t_channel_object_mem
```

holds information about the memory usage for primary data (`usage`) and secondary (derived statistical) data (`usage_2nd`). The logical `lalloc` is `.TRUE.`, if the memory is allocated internally by the CHANNEL interface, and `.FALSE.`, if the memory is pre-allocated externally and specified by the parameter `mem` in subroutine `new_channel_object` (Sect. 5.5.3.1).

The Fortran95 structure `io` stores all information required to control the input / output of the *channel object*:

```
TYPE t_channel_object_io
   !
   ! RESTART HANDLING
   LOGICAL :: lrestart      = .FALSE. ! OUTPUT TO RESTART FILE ?
   LOGICAL :: lignore       = .FALSE. ! IGNORE lrestreq ?
   !
   ! OUTPUT FLAGS
   LOGICAL, DIMENSION(SND_MAXLEN) :: lout = .FALSE.
   ! SPECIAL FOR CONDITIONAL COUNTER (CNT) / AVAERAGE (CAV)
   REAL(DP), DIMENSION(2)         :: range = &
        (/ -HUGE(0.0_DP), HUGE(0.0_DP) /)
   !
END TYPE t_channel_object_io
```

If the logical `lrestart` is `.TRUE.`, the *channel object* is written to the channel specific restart file. With `lignore=.TRUE.` the fact that a *channel object* is usually required for restart (`lrestreq=.TRUE.`) is ignored. This can for instance be used, if a simulation is continued from restart files, but with additional submodels switched on. It can be controlled by the `CTRL`-namelist of CHANNEL (Sect. 2.1). The output request for additional derived statistical data is switched by `lout`; `SND_MAXLEN` is the number of implemented statistics (currently 7: INST, AVE, STD, MIN, MAX, CNT, CAV). The range of valid values (for the conditional counting (CNT) and / or averaging (CAV)) is stored in `range`.

The Fortran95 structure `int` is used to store additional information for internal use only:

```
TYPE t_channel_object_int
   !
   ! OUTPUT AND RESTART
   LOGICAL :: lout = .FALSE.     ! ANY OUTPUT ?
   LOGICAL :: lrst = .FALSE.     ! ANY RESTART ?
   LOGICAL :: lign = .FALSE.     ! IGNORE lrestreq ?
   LOGICAL :: lref = .FALSE.     ! IS reference ?
   ! EXPORT DATA ?
   LOGICAL, DIMENSION(SND_MAXLEN, IOMODE_MAX) :: lexp = .FALSE.
   ! ... MEMORY MANAGEMENT FOR PRIMARY AND SECONDARY DATA
   INTEGER, DIMENSION(SND_MAXLEN) :: i2nd = 0  ! INDEX IN 2ndary DATA
   INTEGER :: n2nd = 0  ! SECONDARY DATA DIMENSION
   !
```

```
    ! MISC
    ! - FIELD HAS BEEN SET FROM RESTART FILE
    LOGICAL, DIMENSION(SND_MAXLEN) :: lrestart_read = .FALSE.
    !
    ! netCDF
    TYPE(t_channel_object_netcdf), DIMENSION(IOMODE_MAX) :: netcdf
    TYPE(t_channel_object_cdi),    DIMENSION(IOMODE_MAX) :: cdi
    !
    ! +++ ADD OTHER OUTPUT FORMATS HERE
    !
  END TYPE t_channel_object_int
```

The logical `lout` is `.TRUE.`, if output of primary or secondary (derived statistical) data is requested, `lrst` controls the output into the *channel* specific restart file. If a *channel object* is requested from a restart file but not present, `lign` is used, if this is to be ignored, e.g., triggered by a respective namelist entry (see Sect. 2.1). The fourth logical, `lref`, is `.TRUE.`, if the *channel object* is a *reference* to another *channel object*.

The two-dimensional logical array `lexp` controls the output of secondary (derived statistical) data. The first dimension indicates the statistical quantity, and the second dimension the output mode. Currently two output modes are implemented (`IOMODE_MAX` = 2) for output and restart, rspectively

The number of secondary data fields for the *channel object*, i.e., the dimension of the `sdat` pointer array (see above), is stored in `n2nd`, whereas the one-dimensional array `i2nd` contains the indices of the corresponding statistical quantities in the `sdat` pointer array. The elements of the one-dimensional logical array `lrestart_read` are set to `.TRUE.`, if the corresponding secondary data have been (re-)initialised from the restart-file.

The Fortran95 structure `netcdf`

```
  ! netCDF I/O (internal use only !)
  TYPE t_channel_object_netcdf
     ! variable ID
     INTEGER                         :: varid        = NC_ID_UNDEF
     ! dimension IDs
     INTEGER                         :: dimid(IRANK) = NC_ID_UNDEF
     ! IDs OF SECONDARY VARIABLES
     INTEGER, DIMENSION(:), POINTER :: svarid => NULL()
  END TYPE t_channel_object_netcdf
```

holds information required for the input / output from / into netCDF files, such as the netCDF variable identifier (`varid`), the corresponding vector with netCDF dimension identifiers (`dimid`), and - if required - the additional netCDF variable identifiers for the secondary derived statistical data (`svarid`).

The Fortran95 structure `cdi`

```
  TYPE t_channel_object_cdi
     ! CDI output?
     LOGICAL                         :: lout    = .FALSE.
     ! variable ID
     INTEGER                         :: varid   = CDI_UNDEFID
     ! grid ID
     INTEGER                         :: gridid  = CDI_UNDEFID
     ! zaxis ID
     INTEGER                         :: zaxisid = CDI_UNDEFID
     ! IDs OF SECONDARY VARIABLES
     INTEGER, DIMENSION(:), POINTER :: svarid => NULL()
  END TYPE t_channel_object_cdi
```

provides information for output via the CDI library. The switch `lout` defines if CDI output is written for the *channel object*. The identifiers for variable (`varid`), grid (`gridid`), and vertical axis (`zaxisid`) are defined during initialisation

and used for the output of variables via the CDI library. Additionally, variable identifiers for secondary derived statistical data are stored, if required.

Similar structures for other input / output formats might be introduced in the same way.

A multitude of *channel objects* contained in a *channel* (Sect. 3.6) is stored as a concatenated list of *channel objects*:

```
TYPE t_channel_object_list
   TYPE(t_channel_object)               :: this
   TYPE(t_channel_object_list), POINTER :: next => NULL()
END TYPE t_channel_object_list
```

Basic subroutines for managing *channel objects* are contained in the Fortran95 module `messy_main_channel.f90` (Sect. 5.5).

3.6 Channels

A *channel* contains a set of *channel objects* and additional meta-information, implemented as a Fortran95 structure:

```
TYPE t_channel
   ! IDENTIFICATION
   CHARACTER(LEN=STRLEN_CHANNEL)    :: name  = ''       ! NAME
   INTEGER                          :: id    = 0        ! ID
   TYPE(t_attribute_list), POINTER  :: att   => NULL()  ! CHANNEL ATTRIBUTES
   TYPE(t_channel_object_mem)       :: memory           ! MEMORY MANAGEMENT
   TYPE(t_channel_io)               :: io               ! I/O
   TYPE(t_channel_def)              :: default          ! OBJECT DEFAULTS
   ! INTERNAL
   TYPE(t_channel_int)              :: int              ! FOR INTERNAL USE
   !
   ! CHANNEL OBJECTS
   TYPE(t_channel_object_list), POINTER :: list => NULL()
END TYPE t_channel
```

The `list` points to the concatenated list of *channel objects*; in empty channels, `list` is `NULL`. Every *channel* is identified by its unique `name` and identifier (`id`), and can have channel specific *attributes*, which are stored in `att` (see Sect. 3.1). The string length `STRLEN_CHANNEL` is `2*STRLEN_SHORT + 1`.

In `memory`, a variable of the type

```
TYPE t_channel_object_mem
   !
   ! MEMORY USAGE
   INTEGER(I8) :: usage       = 0_I8  ! primary data section
   INTEGER(I8) :: usage_2nd   = 0_I8  ! secondary data section
   !
   ! FLAGS FOR INTERNAL USE
   LOGICAL     :: lalloc  = .FALSE.   ! AUTOMATIC MEMORY ALLOCATION
END TYPE t_channel_object_mem
```

contains information about the memory usage of the primary data (`usage`) and the secondary (derived statistical) data (`usage_2nd`). The logical `lalloc` indicates, whether the memory is controlled (allocated / deallocated) internally (`.TRUE.`), or externally (`.FALSE.`) by user-defined memory.

Superordinate input / output related information for all *channel objects* in the *channel* are stored in `io`, a variable of type

```
TYPE t_channel_io
   ! OUTPUT FILE TYPE
   INTEGER, DIMENSION(IOMODE_MAX)  :: ftype  = FTYPE_DEFAULT
   ! NO. OF TIME STEPS PER FILE (OUT)
   INTEGER                         :: ntpf   = 0
   !
END TYPE t_channel_io
```

where `ftype` denotes the file type (format) of each output mode (output or restart), and `ntpf` denotes the number of requested time-steps per output-file. This is one method to control the file size of output files (see Sects. 2.1 and 2.2).

Default values for *channel objects* in a specific *channel* can be pre-defined at *channel* definition, and are stored in `default`, which is of type:

```
TYPE t_channel_def
   !
   ! DEFAULT REPRESENTATION
   INTEGER                    :: reprid   = REPR_UNDEF
   ! DEFAULT: NOT REQUIRED IN RESTART
   LOGICAL                    :: lrestreq = .FALSE.
   ! DEFAULT OUTPUT FLAGS
   TYPE (t_channel_object_io) :: io
   !
END TYPE t_channel_def
```

The identifier of the default *representation* (see Sect. 3.4) is stored in `reprid`. If the *channel objects* need to be saved in the restart file, `lrestreq` is `.TRUE.`. And the default input / output settings are stored in `io`. All *channel* specific defaults can be overwritten by the *channel object* definition.

For internal use only, additional information is stored in `int`:

```
TYPE t_channel_int
   !
   ! OUTPUT AND RESTART
   LOGICAL  :: lout      = .FALSE.         ! OUTPUT ? ANY OBJECT ?
   LOGICAL  :: lrst      = .FALSE.         ! RESTART ? ANY OBJECT ?
   LOGICAL  :: lrestreq  = .FALSE.         ! ANY OBJECT REQUIRED IN RESTART
   LOGICAL  :: lign      = .FALSE.         ! IGNORE ALL (!) lrestreq ?
   ! - OUTPUT FILENAME : <EXPERIMENT (15)>_YYYYMMDD_HHMM_<CHANNEL>.<ext>
   ! - RESTART FILENAME: restart_<CHANNEL>.<ext>
   CHARACTER(LEN=350+STRLEN_CHANNEL+74), DIMENSION(IOMODE_MAX) :: fname = ''
   !
   ! TIMER
   LOGICAL  :: lout_now   = .FALSE.        ! TIME MANAGER
   INTEGER  :: ntpfcnt    = 0              ! COUNTER OF TIME STEPS PER FILE
   LOGICAL  :: lnew_file  = .FALSE.        ! OPEN NEW FILE ?
   REAL(DP) :: tslo       = 0.0_DP         ! time [s] since last output
   ! FORCE OUTPUT (to be set by set_channel_output)
   LOGICAL  :: lforce_out  = .FALSE.
   ! SUPPRESS OUTPUT (to be set by set_channel_output)
   LOGICAL  :: lsuppr_out  = .FALSE.
   ! FORCE NEW FILE
   LOGICAL  :: lforce_newfile = .FALSE.
   !
   LOGICAL :: l2ndreinit1 = .TRUE.
   LOGICAL :: l2ndreinit2 = .FALSE.
   !
   ! netCDF
```

```
    TYPE(t_channel_netcdf), DIMENSION(IOMODE_MAX) :: netcdf
    TYPE(t_channel_cdi), DIMENSION(IOMODE_MAX) :: cdi
    !
    ! +++ ADD OTHER OUTPUT FORMATS HERE
    !
  END TYPE t_channel_int
```

The logicals `lout` and `lrst` are `.TRUE.`, if at least one of the *channel objects* in the *channel* are to be output or saved in the restart file, respectively. A similar flag, `lrestreq`, indicates if at least one of the *channel objects* in the *channel* is required to be re-initialised from the restart file. And `lign` controls, if the restart requirements of all *channel objects* in the *channel* shall be ignored. This is for instance useful, if an additional submodel is switched on after a model simulation starting from restart files.

All *channel objects* in the *channel* are output to one common file, and likewise all *channel objects* in the *channel*, that need to be saved for restart, are saved in one common restart file. The respective filenames (without format specific extension) are stored in `fname` for both modes (output and restart).

The organisation of the channel output within the time loop of a simulation is controlled by several variables: `loutput_now` is `.TRUE.`, if output is active in the present time step. The counter `ntpfcnt` counts the number of output-time steps in the output-file; if `lnew_file` is `.TRUE.`, a new output file (with a new name) is started, instead of appending further output time-steps to an existing file. For the secondary derived statistical analysis, `tslo` stores the simulation time (in seconds) since the last output. The switches `lforce_out` and `lsuppr_out` are used to force or suppress output in a time step, respectively, independent on the output control settings in the namelists (see Sects. 2.1 and 2.2). This is for instance required for specific diagnostic submodels. A similar switch, `lforce_newfile`, is used to force the creation of a new output file (instead of appending data to an existing one). The re-initialisation of the secondary derived data after the output is controlled by `l2ndreinit1` and `l2ndreinit2`.

File format specific information for netCDF files are stored in `netCDF`, another Fortran95 structure,

```
  TYPE t_channel_netcdf
     INTEGER  :: fileID     = NC_ID_UNDEF
     INTEGER  :: dimid_time = NC_ID_UNDEF
  END TYPE t_channel_netcdf
```

which stores the netCDF file identifier (`fileID`) and the netCDF dimension identifier of the time dimension (`dimid_time`). The specific information for CDI is stored in the Fortran95 structure `cdi`,

```
  TYPE t_channel_cdi
     INTEGER  :: fileID     = CDI_UNDEFID
     INTEGER  :: vlistID    = CDI_UNDEFID
     INTEGER  :: taxisID    = CDI_UNDEFID
     INTEGER  :: patch_id   = -1

     INTEGER  :: CellGridID = CDI_UNDEFID
     INTEGER  :: EdgeGridID = CDI_UNDEFID
     INTEGER  :: VertGridID = CDI_UNDEFID

     INTEGER  :: ZaxisID(max_z_axes) = CDI_UNDEFID
  END TYPE t_channel_cdi
```

which stores the CDI specific identifiers for file (`fileID`), variable list (`vlistID`), and time axis (`taxisID`). It also stores on which computational patch the channel is defined (`patch_id`), the identifiers of the horizontal grid points in cells, on edges, and on verticies (`CellGridID`, `EdgeGridID`, and `VertGridID`, respectively) , and the identifier of the vertical axis (`ZaxisID`).

Similar structures for other input / output formats might be introduced in the same way.

Finally, the complete set of channels is stored as a concatenated list of channels:

```
  TYPE t_channel_list
     TYPE(t_channel)               :: this
     TYPE(t_channel_list), POINTER :: next => NULL()
  END TYPE t_channel_list
```

Basic subroutines for managing *channels* are contained in the Fortran95 module `messy_main_channel.f90` (Sect. 5.5).

For models with nested patches and internal coupling, the public variable `current_patch` stores the number of the patch, the model system is currently operating on. This allows for an automatic handling of *channels* and *channel objects* on multiple patches. This is implemented in a way, transparent to existing submodels.

4 Error handling

All subroutines of CHANNEL return the Fortran95 variable `status`, an `INTENT(OUT)` parameter of type `INTEGER`, which indicates a status information of the respective subroutine. The `status` is 0, if the routine was successful, and > 0, if an error occurred. The value of `status` can be transformed into an error message with the function `channel_error_str` in module `messy_main_channel_error.f90`:

`FUNCTION channel_error_str`		`(status)`	
name	type	intent	description
mandatory arguments:			
status	INTEGER	IN	error status
channel_error_str	CHARACTER(LEN=STRLEN_VLONG)	OUT	error message

5 Subroutines for handling the basic entities

5.1 The file `messy_main_channel_attributes.f90`

5.1.1 The subroutine `add_attribute`

`SUBROUTINE add_attribute`		`(status ,list ,name [,i] [,c] [,r] [,loverwrite] [,iflag])`	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
list	TYPE(t_attribute_list)	POINTER	list of attributes
name	CHARACTER(LEN=*)	IN	name of new attribute
optional arguments:			
i	INTEGER	IN	integer value of attribute
c	CHARACTER(LEN=*)	IN	string value of attribute
r	REAL(DP)	IN	real value of attribute
loverwrite	LOGICAL	IN	overwrite if attribute exists?
iflag	INTEGER	IN	check attribute at initialisation

With this subroutine, the *attribute* `name` is added to the *attribute list* `list`. The type of the attribute is specified by value assignment to the corresponding optional argument for INTEGER (`i`), string (CHARACTER, `c`) or REAL(DP) (`r`), respectively. With the optional argument `loverwrite` set to `.TRUE.`, an already existing attribute of the same `name` will be overwritten (`status` is 0), whereas per default (`.FALSE.`) an existing attribute remains untouched and the `status` is > 0.

The optional flag `iflag` controls the specific relevance of the attribute after restart; possible values are:

- AF_NONE (default): no specific relevance,
- AF_RST_CMP: the actual attribute is compared to that from the restart file; differences will cause an error,
- AF_RST_INP: the actual attribute is read from the restart file.

Thus, `iflag` can for instance be used to prevent the usage of restart files, which do not match the actual model setup (e.g., due to a different basemodel resolution).

5.1.2 The subroutine `write_attribute`

SUBROUTINE write_attribute		(status, att | list)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
att*)	TYPE(t_attribute)	IN	attribute
list*)	TYPE(t_attribute_list)	POINTER	list of attributes

*)Note: This subroutine is twofold overloaded for single attributes (`att`) and attribute lists (`list`), respectively.

This subroutine outputs the *attribute* `att` or the *attribute list* `list` to the standard output.

5.1.3 The subroutine `return_attribute`

SUBROUTINE return_attribute		(status, list, name [,i] [,c] [,r] [,iflag])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
list	TYPE(t_attribute_list)	POINTER	list of attributes
name	CHARACTER(LEN=*)	IN	name of attribute
optional arguments:			
i	INTEGER	OUT	integer value of attribute
c	CHARACTER(LEN=*)	OUT	string value of attribute
r	REAL(DP)	OUT	real value of attribute
iflag	INTEGER	OUT	flag for initialisation check

With this subroutine the value of an *attribute* and / or its `iflag` are retrieved. The type is selected by parameter assignment to the corresponding optional argument for INTEGER (`i`), string (CHARACTER, `c`) or REAL(DP) (`r`), respectively.

5.1.4 The subroutine `delete_attribute`

SUBROUTINE delete_attribute		(status, list, name)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
list	TYPE(t_attribute_list)	POINTER	list of attributes
name	CHARACTER(LEN=*)	IN	name of attribute

With this subroutine the *attribute* `name` is removed from the *attribute list* `list`.

5.1.5 The subroutine `copy_attribute_list`

SUBROUTINE copy_attribute_list		(status, list1, list2)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
list1	TYPE(t_attribute_list)	POINTER	source list of attributes
list2	TYPE(t_attribute_list)	POINTER	new list of attributes

With this subroutine a copy of an *attribute list* is constructed.

5.1.6 The subroutine `clean_attribute_list`

SUBROUTINE clean_attribute_list		(status ,list)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
list	TYPE(t_attribute_list)	POINTER	list of attributes

With this subroutine, an *attribute list* is emptied.

5.2 The file messy_main_channel_dimvar.f90

5.2.1 The subroutine add_dimvar

SUBROUTINE add_dimvar		(status, list, name, val)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
list	TYPE(t_dimvar_list)	POINTER	list of dimension variables
name	CHARACTER(LEN=*)	IN	name of dimension variable
val	REAL(DP), DIMENSION(:)	IN	values of dimension variable

This subroutine adds a *dimension variable* to a `list` of dimension variables. The *dimension variable* is identified by its (unique) `name`; the discrete values along the finite axis are specified by a the array `val`.

5.2.2 The subroutine add_dimvar_att

SUBROUTINE add_dimvar_att		(status, list, name, aname [,i] [,c] [,r] [,loverwrite] [,iflag])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
list	TYPE(t_dimvar_list)	POINTER	list of dimension variables
name	CHARACTER(LEN=*)	IN	name of dimension variable
aname	CHARACTER(LEN=*)	IN	name of attribute
optional arguments:			
i	INTEGER	IN	integer value of attribute
c	CHARACTER(LEN=*)	IN	string value of attribute
r	REAL(DP)	IN	real value of attribute
loverwrite	LOGICAL	IN	overwrite if attribute exists?
iflag	INTEGER	IN	check attribute at initialisation

With this subroutine, an *attribute* (`aname`) is specified for the *dimension variable* (`name`). The type of the attribute is specified by value assignment to the corresponding optional argument for INTEGER (`i`), string (CHARACTER, `c`) or REAL(DP) (`r`), respectively. The arguments `loverwrite` and `iflag` are the same as in `SUBROUTINE add_attribute` (Sect. 5.1.1).

5.2.3 The subroutine write_dimvar

SUBROUTINE write_dimvar		(status , dimvar | list)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
dimvar*)	TYPE(t_dimvar)	IN	dimension variable
list*)	TYPE(t_dimvar_list)	POINTER	list of dimension variables

*)Note: This subroutine is twofold overloaded for writing a specific dimension variable (`dimvar`) or a complete list of dimension variables (`list`), respectively.

This subroutine outputs information on the *dimension variable* `dimvar` or on the list of *dimension variables* `list` to the standard output.

5.2.4 The subroutine get_dimvar

SUBROUTINE get_dimvar		(status, list, name, dimvar)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
list	TYPE(t_dimvar_list)	POINTER	list of dimension variables
name	CHARACTER(LEN=*)	IN	name of dimension variable
dimvar	TYPE(t_dimvar)	POINTER	dimension variable

This subroutine selects a single *dimension variable* specified by its `name` from a `list` of *dimension variables*.

5.2.5 The subroutine `delete_dimvar`

SUBROUTINE delete_dimvar		(status, list, name)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
list	TYPE(t_dimvar_list)	POINTER	list of dimension variables
name	CHARACTER(LEN=*)	IN	name of dimension variable

This subroutine removes a *dimension variable* specified by its `name` from a `list` of *dimension variables*.

5.2.6 The subroutine `clean_dimvar_list`

SUBROUTINE clean_dimvar_list		(status, list)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
list	TYPE(t_dimvar_list)	POINTER	list of dimension variables

This subroutine empties a `list` of *dimension variables*.

5.3 The file `messy_main_channel_dimensions.f90`

5.3.1 The subroutine `new_dimension`

SUBROUTINE new_dimension		(status, id, name, len [,ltime] [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
id	INTEGER	OUT	dimension identifier
name	CHARACTER(LEN=*)	IN	name of dimension
len	INTEGER	IN	dimension length
optional arguments:			
ltime	LOGICAL	IN	dimension is time?
patch_id	INTEGER	IN	number of patch

With this subroutine, a new *dimension* is added to the central list of dimensions. The `name` of the dimension must be unique, the dimension identifier (`id`) is internally set and returned for later reference. The length of the *dimension* is specified by `len`, and the optional parameter `ltime` (default: `.FALSE.`) must be `.TRUE.` for the time dimension. For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` = 1.

5.3.2 The subroutine `add_dimension_variable`

SUBROUTINE add_dimension_variable		(status, dname | id, vname, val [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
dname*)	CHARACTER(LEN=*)	IN	name of dimension
id*)	INTEGER	IN	dimension identifier
vname	CHARACTER(LEN=*)	IN	name of dimension variable
val	REAL(DP), DIMENSION(:)	IN	values of dimension variable
optional arguments:			
patch_id	INTEGER	IN	number of patch

*) Note: This subroutine is twofold overloaded for usage of the *dimension* name or the *dimension* identifier, respectively.

This subroutine adds a *dimension variable* specified by its name (`vname`) to a *dimension*. The latter is either specified by its name (`dname`) or its identifier (`id`). The discrete, finite axes values are specified by the array `val`, which must have the dimension length. For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` = 1.

5.3.3 The subroutine `update_dimension_variable`

`SUBROUTINE update_dimension_variable`		`(status, dname, vname, val [,patch_id])`	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
dname	CHARACTER(LEN=*)	IN	name of dimension
vname	CHARACTER(LEN=*)	IN	name of dimension variable
val	REAL(DP), DIMENSION(:)	IN	values of dimension variable
optional arguments:			
patch_id	INTEGER	IN	number of patch

This subroutine updates the axis values (array `val`) of the *dimension variable* `vname` of *dimension* `dname`. This is for instance used to set the actual time axis value (dimension of length 1) within the time integration. For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` = 1.

5.3.4 The subroutine `add_dimension_variable_att`

`SUBROUTINE add_dimensionvariable_att`		`(status, dname | id, vname, aname [,i] [,c] [,r] [,loverwrite] [,iflag] [,patch_id]))`	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
dname*)	CHARACTER(LEN=*)	IN	name of dimension
id*)	INTEGER	IN	dimension identifier
vname	CHARACTER(LEN=*)	IN	name of dimension variable
aname	CHARACTER(LEN=*)	IN	name of attribute
optional arguments:			
i	INTEGER	IN	integer value of attribute
c	CHARACTER(LEN=*)	IN	string value of attribute
r	REAL(DP)	IN	real value of attribute
loverwrite	LOGICAL	IN	overwrite if attribute exists?
iflag	INTEGER	IN	check attribute at initialisation
patch_id	INTEGER	IN	number of patch

*)Note: This subroutine is twofold overloaded for usage of the *dimension* name or the *dimension* identifier, respectively.

With this subroutine, the *attribute* `aname` is added to the *dimension variable* `vname` of a dimension. The latter is either specified by its name (`dname`) or its identifier (`id`). The type of the *attribute* is specified by value assignment to the corresponding optional argument for INTEGER (`i`), string (CHARACTER, `c`) or REAL(DP) (`r`), respectively. The arguments `loverwrite` and `iflag` are the same as in `SUBROUTINE add_attribute` (Sect. 5.1.1). For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` = 1.

5.3.5 The subroutine `get_dimension`

`SUBROUTINE get_dimension`		`(status ,name | id ,dim [,patch_id])`	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
name*)	CHARACTER(LEN=*)	IN	name of dimension
id*)	INTEGER	IN	dimension identifier
dim	TYPE(t_dimension)	POINTER	dimension
optional arguments:			
patch_id	INTEGER	IN	number of patch

*)Note: This subroutine is twofold overloaded for usage of the *dimension* name or the *dimension* identifier, respectively.

This subroutine retrieves a *dimension* from the internal list. The *dimension* is either specified by its `name`, or by its identifier (`id`). For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` = 1.

5.3.6 The subroutine `get_dimension_info`

`SUBROUTINE get_dimension_info`		`(status ,name | dimid [,id | name] [,len] [,patch_id])`	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
name*)	CHARACTER(LEN=*)	IN	name of dimension
dimid*)	INTEGER	IN	dimension identifier
optional arguments:			
id*)	INTEGER	OUT	dimension identifier
name*)	CHARACTER(LEN=*)	IN	name of dimension
len	INTEGER	OUT	dimension length
patch_id	INTEGER	IN	number of patch

*)Note: This subroutine is twofold overloaded for usage of the *dimension* name or the *dimension* identifier, respectively. If the *dimension* name is specified, the *dimension* identifier can be retrieved and vice versa.

This subroutine is used to retrieve information (the identifier `id` (or the name `name`) and / or the lenght `len`) about a *dimension*, which is specified by its `name` (or its identifier `dimid`). For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` = 1.

5.3.7 The subroutine `write_dimension`

`SUBROUTINE write_dimension`		`(status [,dim] [,patch_id])`	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
optional arguments:			
dim*)	TYPE(t_dimension)	IN	dimension
patch_id*)	INTEGER	IN	number of patch

*)Note: This subroutine is twofold overloaded. If no specific *dimension* (`dim`) is given, all *dimensions* are written. `patch_id` is only available if no `dim` is given.

This subroutine outputs information on all *dimensions* (no 2nd argument), or on a specific dimension `dim` to the standard output. In case all *dimensions* are written, the `patch_id` has to be set for models with nested patches and internal coupling, otherwise `patch_id` = 1.

5.3.8 The subroutine `clean_dimensions`

`SUBROUTINE clean_dimensions`		`(status [,patch_id])`	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
optional arguments:			
patch_id*)	INTEGER	IN	number of patch

This subroutine empties the internal list of dimensions. It is usually called only once during the finalising phase of the model. For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` = 1.

5.4 The file messy_main_channel_repr.f90

5.4.1 The subroutine new_representation

SUBROUTINE new_representation			(status, id, name, rank, link, dctype, dimension_ids, ldimlen [,output_order] [,axis] [,nbounds] [,patch_id] [,hgrid] [,vgrid])
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
id	INTEGER	OUT	representation identifier
name	CHARACTER(LEN=*)	IN	name of representation
rank	INTEGER	IN	rank of representation
link	CHARACTER(LEN=IRANK)	IN	rank mapping
dctype	INTEGER	IN	type of parallel decomposition
dimension_ids	INTEGER, DIMENSION(rank)	IN	vector of dimension identifiers
ldimlen	INTEGER, DIMENSION(rank)	IN	local (decomposed) dimension lengths
optional arguments:			
output_order	INTEGER, DIMENSION(rank)	IN	output order of ranks
axis	CHARACTER(LEN=IRANK)	IN	geometry information
nbounds	INTEGER, DIMENSION(rank)	IN	number of boundary points
patch_id	INTEGER	IN	number of patch
hgrid	INTEGER	IN	horizontal grid id
vgrid	INTEGER	IN	vertical grid id

This subroutine defines a new *representation*. The *representation* must have a unique `name`, its identifier (`id`) is returned for later reference. The `rank` of the *representation* can be $0 \leq$ `rank`$\leq$`IRANK`, where `IRANK` = 4 in the current implementation. The `link` specifies which of the internal IRANK ranks are used ('x') or unused ('-'), `dimension_ids` is a vector (of rank `rank`) with the *dimension* identifiers of the dimensions that span the representation.

In parallel environments, the vector `ldimlen` specifies the 'local' dimension lengths on the actual parallel process, and the integer `dctype` is used to select the corresponding gather and scatter subroutines for the parallel re- and decomposition of the data, respectively. The pre-defined parameter `AUTO` can be used as element of the `ldimlen` vector to indicate that no parallel decomposition along the corresponding dimension (rank) is implemented. In that case, the 'local' dimension length equals the 'global' (i.e., the original) dimension length.

The optional vector `output_order` can be used to re-order the IRANK ranks for the output of data in this representation.

The optional string `axis` is used to specify the geometric meaning of the corresponding ranks, i.e. 'X', 'Y', and 'Z' for the spatial dimensions, 'N' for index axes, and '-', if the axis is not further specified. This information can be used for generic transformation and transposition routines. The default is '----'.

The optional vector `nbounds` is used to specify the number of additional 'local' (i.e., parallel decomposed) boundary points along each dimension at both ends, which implies that for all *channel objects* in this represention, the 'local' (i.e., parallel decomposed) data array is increased by 2× `nbounds(i)` in the direction of dimension `i`. The default is zero additional boundary points for all dimensions.

For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise the representations `patch_id` is undefined (`patch_id` = -1). The new `representation` is valid for the specified `patch`, so the `name` has not to be unique over different `patches`, the returned `id` is unique for every `representation`.

For CDI, the optional parameters `hgrid` and `vgrid` set the identifiers of the horizontal and vertical grid, respectively.

5.4.2 The subroutine `set_representation_decomp`

`SUBROUTINE set_representation_decomp`		`(status ,id ,start ,cnt ,mu ,ml [,lchk] [,piotype])`	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
id	INTEGER	IN	representation identifier
start	INTEGER, DIMENSION(:,:)	IN	start vector in global index range
cnt	INTEGER, DIMENSION(:,:)	IN	count vector in global index range
mu	INTEGER, DIMENSION(:,:)	IN	memory lower bound
ml	INTEGER, DIMENSION(:,:)	IN	memory upper bound
optional arguments:			
lchk	LOGICAL	IN	check memory size?
piotype	INTEGER	IN	type of parallel input / output

With this subroutine the parallel decomposition of a representation, specified by its identifier (`id`), is set for parallel input / output. Currently only parallel netCDF is implemented.

The integer arrays `start`, `cnt`, `mu` and `ml` have the dimension `nseg` $\times$ `IRANK`, where `nseg` is the number of consecutive segments in the global (output) field and `IRANK` (=4) is the internal rank of the data.

For a given segment (1st index), `start` specifies the start indices in the global (output) array, `cnt` the number of steps (from `start`) in the global (output) array, and `mu` and `ml` the upper and lower index bounds in the 'local' (i.e., decomposed on the actual parallel process) memory array, respectively.

The optional parameter `lchk` (default: `.TRUE.`) is used to switch off (`.FALSE.`) an internal consistency check between the local (decomposed) and global (output) array lengths. In case the arrays are internally further reshaped (e.g., for vectorisation), this internal check must be switched off.

The optional parameter `piotype` is used to control the parallel input / output mode:

- PIOTYPE_SGL: output of a single parallel process,
- PIOTYPE_COL: collective output of all parallel processes,
- PIOTYPE_IND: independent output of all parallel processes.

5.4.3 The subroutine `write_representation`

`SUBROUTINE write_representation`		`(status [,repr])`	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
repr*)	TYPE(t_representation)	IN	representation

*)Note: This subroutine is twofold overloaded for writing a specific *representation* (`repr`) or all *representations*, respectively.

This subroutine outputs information on all *representations* (no 2nd argument), or on a specific representation `repr` to the standard output.

5.4.4 The subroutine `write_representation_dc`

`SUBROUTINE write_representation_dc`		`(status, p_pe)`	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
p_pe	INTEGER	IN	parallel process identifier

This subroutine outputs the parallel decomposition tables of all *representations* on parallel process `p_pe`.

5.4.5 The subroutine `get_representation`

SUBROUTINE get_representation		(status, name/id, repr [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
name*)	CHARACTER(LEN=*)	IN	name of representation
id*)	INTEGER	IN	representation identifier
repr	TYPE(t_representation)	POINTER	representation
optional arguments:			
patch_id*)	INTEGER	IN	number of patch

*)Note: This subroutine is twofold overloaded for usage of the *representation* `name` or the *representation* identifier (`id`), respectively. `patch_id` is only available, if `name` is given.

This subroutine sets a pointer (`repr`) to a specific *representation*, which is either specified by its `name` or its identifier (`id`). In case `name` is used, the `patch_id` has to be set for models with nested patches and internal coupling.

5.4.6 The subroutine `get_representation_info`

SUBROUTINE get_representation_info		(status, inpname [,id] [,rank] [,link] [,axis] [,gdimlen] [,ldimlen] [,dctype] [,name] [,nbounds] [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
inpname	CHARACTER(LEN=*)	IN	name of representation
optional arguments:			
id	INTEGER	INOUT	representation identifier
rank	INTEGER	OUT	rank of representation
link	CHARACTER(LEN=IRANK)	OUT	rank mapping
axis	CHARACTER(LEN=IRANK)	OUT	geometry information
gdimlen	INTEGER, DIMENSION(IRANK)	OUT	global dimension lengths
ldimlen	INTEGER, DIMENSION(IRANK)	OUT	local (decomposed) dimension lengths
dctype	INTEGER	OUT	type of parallel decomposition
name	CHARACTER(LEN=STRLEN_MEDIUM)	OUT	name of representation
nbounds	INTEGER, DIMENSION(IRANK)	OUT	number of boundary points
patch_id	INTEGER	IN	number of patch

This subroutine is used to retrieve information about a *representation*, which is either identified by its *name*, or by an empty string for its `name` and its identifier (`id`). The optional parameters correspond to those in subroutine `new_representation` (Sect. 5.4.1). In case `name` is used, the `patch_id` has to be set for models with nested patches and internal coupling.

5.4.7 The subroutine `get_representation_id`

SUBROUTINE get_representation_id		(status, name, reprid [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
name	CHARACTER(LEN=*)	IN	name of representation
reprid	INTEGER	OUT	representation identifier
optional arguments:			
patch_id	INTEGER	IN	number of patch

This subroutine is to retrieve the representation identifier (`reprid`) for a given representation specified by its `name`. For models with nested patches and internal coupling, the `patch_id` has to be set.

5.4.8 The subroutine `clean_representations`

SUBROUTINE clean_representations		(status)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	

This subroutine empties the internal list of representations. It is usually called only once during the finalising phase of the model.

5.4.9 The subroutine `repr_reorder`

SUBROUTINE repr_reorder		(status, flag, lparallel, repr, mem, out)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
flag	INTEGER	IN	direction flag
lparallel	LOGICAL	IN	parallel input / output?
repr	TYPE(t_representation)	POINTER	representation
mem	REAL(DP), DIMENSION(:,:,:,:)	POINTER	memory data array
out	REAL(DP), DIMENSION(:,:,:,:)	POINTER	output data array

With this subroutine the shape (rank order) of a data array can be converted from memory layout (`mem`) to output layout (`out`, `flag=1`) or from output / restart file layout (`out`) to memory layout (`mem`, `flag=-1`). `repr` is the pointer to the corresponding *representation*. For `lparallel=.TRUE.` the data array dimension lengths in parallel decomposition (i.e., local) are used (parallel input / output), for `lparallel=.FALSE.` the global array dimensions are used (for serial input / output after gathering the data to a dedicated process).

5.4.10 The subroutine `repr_getptr`

SUBROUTINE repr_getptr		(status, repr, in [,p0] [,p1] [,p2] [,p3] [,p4] [,linner])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
in	REAL(DP), DIMENSION(:,:,:,:)	POINTER	data array
repr	TYPE(t_representation)	POINTER	representation
optional arguments:			
p0	REAL(DP)	POINTER	pointer to data array
p1	REAL(DP), DIMENSION(:),	POINTER	pointer to data array
p2	REAL(DP), DIMENSION(:,:),	POINTER	pointer to data array
p3	REAL(DP), DIMENSION(:,:,:),	POINTER	pointer to data array
p4	REAL(DP), DIMENSION(:,:,:,:)	POINTER	pointer to data array
linner	LOGICAL	IN	select inner part only?

This subroutine sets pointers of rank 0 (`p0`) to IRANK (`p4`) to the data memory (`in`), according to the parameter `link` (see Sect. 5.4.1) of the corresponding *representation* `repr`.

If `linner` is `.TRUE.` (default is `.FALSE.`) and the *representation* is defined with additional boundaries (see Sect. 3.4), the resulting `p0` ... `p4` point to the inner part of the internal data memory, i.e., without the additional boundaries.

5.5 The file `messy_main_channel.f90`

The subroutines and functions in this file constitute the main interface subroutines for the application of CHANNEL from within a model. They are divided into two groups: the first group (to be called from the basemodel interface layer (BMIL)) to provide the overall framework, and the second group (to be called from the submodel interface layer (SMIL)) of MESSy submodels to handle individual *channels* and *objects*. For models with nested patches and internal coupling, a *channel* is unambiguously determined by its *name* and the number of the *patch* (set by `patch_id`

in the subroutines). The public variable `current_patch` is implemented for models with nested patches and internal coupling. It stores the number of the *patch*, the model is currently operating on. For models without nested *patches* `current_patch` = 1.

5.5.1 BMIL subroutines

5.5.1.1 The subroutine `main_channel_read_ctrl`

SUBROUTINE main_channel_read_nml_ctrl		(status ,iou)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
iou	INTEGER	IN	input / output unit

This subroutine is called once during the initialisation phase of the model to read the `CTRL` namelist (see Sect. 2.1).

5.5.1.2 The subroutine `fixate_channels`

SUBROUTINE fixate_channels		(status [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
optional arguments:			
patch_id	INTEGER	IN	number of patch

This subroutine is called once at the end of the initialisation phase of the model. All settings for *channels* and *channel objects* are fixated and after the call to this subroutine no more *channels* or *channel objects* can be created. With the call to this subroutine, the channel based memory layout of the model is complete. For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` is set to `current_patch`. In case of multiple nested patches, this routine is called once for every patch during the initialisation phase.

5.5.1.3 The subroutine `trigger_channel_output`

SUBROUTINE trigger_channel_output		(status, lnow, ltnf, lforce_new [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
lnow	LOGICAL,DIMENSION(:)	IN	trigger channel specific output now
ltnf	LOGICAL,DIMENSION(:)	IN	trigger channel specific new file now
lforce_new	LOGICAL	IN	trigger new files for all channels now
optional arguments:			
patch_id	INTEGER	IN	number of patch

This subroutine is called once within the time loop of the model in order to trigger the output of the *channel objects* in each *channel* (`lnow`), and / or to force the creation of new output files (`ltnf`) instead of appending the output to existing files (for time series). The array lengths of `lnow` and `ltnf` equal the number of *channels*, i.e. output and file creation are controlled for each *channel* individually. With `lforce_new = .TRUE.` new files are created for all *channels*.

For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` is set to `current_patch`. In case of multiple nested patches, this routine is called once for every patch within the time loop.

5.5.1.4 The subroutine `update_channels`

SUBROUTINE update_channels		(status ,flag ,dtime [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
flag	INTEGER	IN	flag for different entry points
dtime	REAL(DP)	IN	time step length
optional arguments:			
patch_id	INTEGER	IN	number of patch

This subroutine is called once within the time loop of the model to update the internal counters (e.g., the output time steps per output file) and to update the internal secondary (statistical) data fields (see Sect. 2.1). The time step length `dtime` is internally used for calculating the statistics (e.g., the average between two output time steps, see Sect. 2.1).

For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` is set to `current_patch`. In case of multiple nested patches, this routine is called once for every patch within the time loop.

5.5.1.5 The subroutine `clean_channels`

SUBROUTINE clean_channels		(status [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
optional arguments:			
patch_id	INTEGER	IN	number of patch

This subroutines empties the complete list of channels and releases all memory consumed by *channels* and *channel objects*, except for the externally pre-allocated memory used with the `mem` option (see Sect. 5.5.3.1). It is usually called once during the finalising phase of the model.

For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` is set to `current_patch`. In case of multiple nested patches, this routine is called once for every patch during the finalising phase of the model.

5.5.2 SMIL subroutines for channels

5.5.2.1 The subroutine `new_channel`

SUBROUTINE new_channel		(status, cname [,reprid] [,lrestreq] [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
cname	CHARACTER(LEN=*)	IN	name of channel
optional arguments:			
reprid	INTEGER	IN	identifier of default representation
lrestreq	LOGICAL	IN	default switch for restart file dump
patch_id	INTEGER	IN	number of patch

This subroutine defines a new *channel* with the unique name `cname`. With the optional parameter `reprid` the default *representation*, specified by its identifier, for all *channel objects* of this *channel* can be pre-set. This can still be overwritten during the creation of the *channel object*, however (see Sect. 5.5.3.1). Likewise, with the optional parameter `lrestreq=.TRUE.` (default is `.FALSE.`) all *channel objects* are written to the *channel* restart file, unless overwritten by creation of the *channel object* (see Sect. 5.5.3.1).

For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` is set to `current_patch`. The `cname` has to be unique in each patch, during channel output the `patch_id` is appended to the output file automatically to avoid collision of channel names.

5.5.2.2 The subroutine write_channel

SUBROUTINE write_channel		(status [,channel | cname] [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
optional arguments:			
channel*)	TYPE(t_channel)	POINTER	channel
cname*)	CHARACTER(LEN=*)	IN	name of channel
patch_id*)	INTEGER	IN	number of patch

*)Note: This subroutine is threefold overloaded to write one specific *channel*, either specified directly (`channel`) or by the *channel* name (`cname`), or to write all *channels* (no second argument), respectively. `patch_id` is only available in cases where `channel` is not specified.

This subroutine outputs summary information about one *channel*, either specified directly or by its name (`cname`), or about all *channels* (no 2nd argument). In case the channel is specified by its name (`cname`) or all channels are selected, the `patch_id` has to be set for models with nested patches and internal coupling, otherwise `patch_id` is set to `current_patch`.

5.5.2.3 The subroutine get_channel_info

SUBROUTINE get_channel_info		(status, cname [,ldims] [,lreprs] [,onames] [,pick] [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
cname	CHARACTER(LEN=*)	IN	name of channel
optional arguments:			
ldims	LOGICAL, DIMENSION(:)	POINTER	dimension flags
lreprs	LOGICAL, DIMENSION(:)	POINTER	representation flags
onames	CHARACTER(LEN=STRLEN_OBJECT)	POINTER	object names
pick	CHARACTER(LEN=*)	IN	selector (restart, output, all)
patch_id	INTEGER	IN	number of patch

With this subroutine, specific information about a *channel* (specified by its name `cname`) can be optionally retrieved: For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` is set to `current_patch`. The length of `ldims` is the number of defined *dimensions* and its elements (index = *dimension* identifier) indicate, if the corresponding *dimension* is used (`.TRUE.`) by at least one *channel object* in the *channel* or not (`.FALSE.`). Likewise, the length of `lreprs` is the number of defined *representations* and its elements (index = *representation* identifier) indicate, if at least one *channel object* in the *channel* is of the corresponding *representation* (`.TRUE.`), or not (`.FALSE.`). The list `onames` contains the names of the *channel objects* in the *channel*.

With `pick` a subset of *channel objects* in the *channel* can be selected for `ldims`, `lreprs` and `onames`:

- 'all': all *channel objects* are selected,
- 'output': only *channel objects* which are output are selected,
- 'restart': only *channel objects* which are written to the restart file are selected.

5.5.2.4 The subroutine get_channel_name

SUBROUTINE get_channel_name		(status, id, cname [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
id	INTEGER	IN	channel identifier
cname	CHARACTER(LEN=STRLEN_CHANNEL)	OUT	name of channel
optional arguments:			
patch_id	INTEGER	IN	number of patch

This subroutine is to retrieve the name (cname) of a *channel* for a given *channel* identifier (id). For models with nested patches and internal coupling, the patch_id has to be set, otherwise patch_id is set to current_patch. *Channel* identifiers (ids) are unique per *patch*.

5.5.2.5 The subroutine set_channel_output

SUBROUTINE set_channel_output		(status, cname, lout [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
cname	CHARACTER(LEN=*)	IN	name of channel
lout	LOGICAL	IN	trigger / suppress output?
optional arguments:			
patch_id	INTEGER	IN	number of patch

With this subroutine called within the time loop of the model, the output of a specific channel, specified by its name (cname), can be forced (lout=.TRUE.) or suppressed (lout=.FALSE.), independent of the corresponding namelist entries (see Sects. 2.1 and 2.2). This is useful for specific diagnostic submodels. For models with nested patches and internal coupling, the patch_id has to be set, otherwise patch_id is set to current_patch.

5.5.2.6 The subroutine set_channel_newfile

subroutine set_channel_newfile		(status, cname, lnew [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
cname	CHARACTER(LEN=*)	IN	name of channel
lnew	LOGICAL	IN	trigger new file?
optional arguments:			
patch_id	INTEGER	IN	number of patch

With this subroutine called within the time loop of the model, the generation of a new output file for the *channel* with name cname can be forced (lnew=.TRUE.), independent of the corresponding namelist entries (see Sects. 2.1 and 2.2). This is useful for specific diagnostic submodels. For models with nested patches and internal coupling, the patch_id has to be set, otherwise patch_id is set to current_patch.

5.5.3 SMIL subroutines for channel objects

5.5.3.1 The subroutine new_channel_object

SUBROUTINE new_channel_object		(status ,cname ,oname [,p0] [,p1] [,p2] [,p3] [,p4] [,mem] [,reprid] [,lrestreq] [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
cname	CHARACTER(LEN=*)	IN	name of channel
oname	CHARACTER(LEN=*)	IN	name of object
optional arguments:			
p0	REAL(DP)	POINTER	pointer to data array
p1	REAL(DP), DIMENSION(:)	POINTER	pointer to data array
p2	REAL(DP), DIMENSION(:,:)	POINTER	pointer to data array
p3	REAL(DP), DIMENSION(:,:,:)	POINTER	pointer to data array
p4	REAL(DP), DIMENSION(:,:,:,:)	POINTER	pointer to data array
mem	REAL(DP), DIMENSION(:,:,:,:)	POINTER	external data array
reprid	INTEGER	IN	representation identifier
lrestreq	LOGICAL	IN	dump to restart file?
patch_id	INTEGER	IN	number of patch

This subroutine defines a new *channel object* in an existing *channel*. The name of the *channel* is `cname` and the name of the *channel object* is `oname` and must be unique within the *channel*. The optional pointers `p0`, `p1`, `p2`, `p3` and `p4` of rank 0 to IRANK=4, respectively, point to the internal data memory, according to the *representation* (see parameter `link`, Sect. 5.4.1) of the *channel object*. The *representation* of the *channel object* is defined by either the default (see Sect. 5.5.2.1) or by the parameter `reprid`, in both cases by the *representation* identifier.

The parameter *lrestreq*, of which the default for all *objects* in one *channel* can be pre-set according to Sect. 5.5.2.1, controls if the corresponding *channel object* needs to be written to the restart file (`.TRUE.`) or not (`.FALSE.`).

With the optional pointer `mem` (of rank IRANK=4), specific, pre-allocated memory for the *channel object* can be specified. In this case, the internal automatic memory management is bypassed and the user must take care of the correct allocation and deallocation of the memory!

For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` is set to `current_patch`.

5.5.3.2 The subroutine `get_channel_object`

`SUBROUTINE get_channel_object`		`(status, cname, oname [,p0] [,p1] [,p2] [,p3] [,p4], [,linner] [,patch_id])`	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
cname	CHARACTER(LEN=*)	IN	name of channel
oname	CHARACTER(LEN=*)	IN	name of object
optional arguments:			
p0	REAL(DP)	POINTER	pointer to data array
p1	REAL(DP), DIMENSION(:)	POINTER	pointer to data array
p2	REAL(DP), DIMENSION(:,:)	POINTER	pointer to data array
p3	REAL(DP), DIMENSION(:,:,:)	POINTER	pointer to data array
p4	REAL(DP), DIMENSION(:,:,:,:)	POINTER	pointer to data array
linner	LOGICAL	IN	select inner part only?
patch_id	INTEGER	IN	number of patch

This subroutine is used to set pointers of rank 0 to IRANK=4 (`p0` ... `p4`) to the internal data memory of the *channel object* `oname` in *channel* `cname`, according to the *representation* of the *channel object* (see parameter `link`, Sect. 5.4.1).

If `linner` is `.TRUE.` (default is `.FALSE.`) and the *representation* is defined with additional boundaries (see Sect. 3.4), the resulting `p0` ... `p4` point to the inner part of the internal data memory, i.e., without the additional boundaries.

For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` is set to `current_patch`.

5.5.3.3 The subroutine `get_channel_object_info`

`SUBROUTINE get_channel_object_info`		`(status, cname, oname [,lrestart_read] [,reprid] [,axis] [,nbounds] [,patch_id])`	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
cname	CHARACTER(LEN=*)	IN	name of channel
oname	CHARACTER(LEN=*)	IN	name of object
optional arguments:			
lrestart_read	LOGICAL	OUT	flag to read from restart file
reprid	INTEGER	OUT	representation identifier
axis	CHARACTER(LEN=IRANK)	OUT	geometric information
nbounds	INTEGER, DIMENSION(IRANK)	OUT	number of boundary points
patch_id	INTEGER	IN	number of patch

This subroutine is used to retrieve information about a specific *channel object* (`oname`) in *channel* `cname`, namely if it was read from the corresponding restart file (then `lrestart_read=.TRUE.`) and its *representation* identifier. Moreover,

additional geometric information about the underlying *representation* can be retrieved (see Sect. 3.4): the axes (with `axis`) and the number of additional boundary points in each dimension (`nbounds`).

For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` is set to `current_patch`.

5.5.3.4 The subroutine `new_channel_object_reference`

`SUBROUTINE new_channel_object_reference`		`(status, cname1, oname1, cname2, oname2 [,lcopyatt] [,patch_id])`	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
cname1	CHARACTER(LEN=*)	IN	name of existing channel
oname1	CHARACTER(LEN=*)	IN	name of existing object
cname2	CHARACTER(LEN=*)	IN	name of channel to add reference to
oname2	CHARACTER(LEN=*)	IN	name of object in referencing channel
optional arguments:			
lcopyatt	LOGICAL	IN	copy all object attributes?
patch_id	INTEGER	IN	number of patch

This subroutine creates a reference (with name `oname2`) in a second *channel* (`cname2`) to a *channel object* (`oname1`) in channel `cname1`. The primary data memory is shared between the original object and its reference, the secondary data memory (for the statistical analyses) is separately allocated (see Sect. 2.1), depending on the requests specified in the `CTRL`-namelist (see Sect. 2.1).

For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` is set to `current_patch`.

5.5.3.5 The subroutine `set_channel_object_restreq`

`SUBROUTINE set_channel_object_restreq`		`(status, cname, oname [,patch_id])`	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
cname	CHARACTER(LEN=*)	IN	name of channel
oname	CHARACTER(LEN=*)	IN	name of object
optional arguments:			
patch_id	INTEGER	IN	number of patch

This subroutine is used to set the restart flag of a specific *channel object* (`oname`) in *channel* `cname`, in order to force the output of the object into the corresponding restart file.

For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` is set to `current_patch`.

5.5.3.6 The subroutine `get_channel_object_dimvar`

SUBROUTINE get_channel_object_dimvar		(status, cname, oname, dva [, units] [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
cname	CHARACTER(LEN=*)	IN	name of channel
oname	CHARACTER(LEN=*)	IN	name of object
dva	TYPE (PTR_1D_ARRAY), DIMENSION(:)	POINTER	pointer to dimension
	DIMENSION(:)		variable data array
optional arguments:			
units	CHARACTER(LEN=STRLEN_ULONG),	POINTER	units of
	DIMENSION(:)		dimension variables
patch_id	INTEGER	IN	number of patch

This subroutine is used to access the *dimension variables* of the *channel object* (`oname`) in *channel* `cname`. The values of the dimension variables are accessed through the pointer `dva`, the corresponding "units" *attributes* are optionally retrieved with the string array pointer `units`. For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` is set to `current_patch`.

The parameter `dva` is of type

```
TYPE PTR_1D_ARRAY
   REAL(DP), DIMENSION(:), POINTER :: PTR
END TYPE PTR_1D_ARRAY
```

Example: The values of the 2nd dimension variable of the *channel object* are accessed by `dva(2)%ptr(:)`, the corresponding unit is `units(2)`. Both, `dva` and `units` are allocated according to the number of dimensions of the *channel objects representation*. In case a *dimension* has more than one corresponding *dimension variable*, the first is used. If no "units" *attribute* is defined for *dimension variable* of *dimension* n, `units(n)` contains an empty string.

5.5.4 SMIL subroutines for attributes

5.5.4.1 The subroutine `new_attribute`

SUBROUTINE new_attribute		(status, (list, name) | (ganame) | (cname, caname) | (cname, oname, oaname) [,i] [,c] [,r] [,loverwrite] [,iflag] [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
list*)	TYPE(t_attribute_list)	POINTER	list of attributes
name*)	CHARACTER(LEN=*)	IN	name of new attribute
ganame*)	CHARACTER(LEN=*)	IN	name of new global attribute
cname*)	CHARACTER(LEN=*)	IN	name of channel
caname*)	CHARACTER(LEN=*)	IN	name of new channel attribute
cname*)	CHARACTER(LEN=*)	IN	name of channel
oname*)	CHARACTER(LEN=*)	IN	name of object
oaname*)	CHARACTER(LEN=*)	IN	name of new channel object attribute
optional arguments:			
i	INTEGER	IN	integer value of attribute
c	CHARACTER(LEN=*)	IN	string value of attribute
r	REAL(DP)	IN	real value of attribute
loverwrite	LOGICAL	IN	overwrite if attribute exists?
iflag	INTEGER	IN	check attribute at initialisation
patch_id*)	INTEGER	IN	number of patch

*)Note: This subroutine is fourfold overloaded to add an *attribute* (`name`) to an *attribute list* (`list`), to add a *global attribute* (`ganame`), to add a *channel attribute* (`cname` is the *channel* name and `caname` the *channel attribute* name),

or to add a *channel object attribute* (`cname` is the *channel* name, `oname` the *channel object* name, `oaname` the *channel object attribute* name), respectively. `patch_id` is only available, when adding a *channel attribute* or *channel object attribute*.

This subroutine provides a comfortable interface to define *attributes* for various entities, namely arbitrary *attribute lists* (`list` and `name`), centrally stored *global attributes* (`ganame`), *channel attributes* (`cname` and `caname`) and *channel object attributes* (`cname`, `oname` and `oaname`).

The type of the attribute is in all cases specified by value assignment to the corresponding optional argument for INTEGER (`i`), string (CHARACTER, `c`) or REAL(DP) (`r`), respectively. The arguments `loverwrite` and `iflag` are the same as in `SUBROUTINE add_attribute` (Sect. 5.1.1).

When adding a *channel attribute* or *channel object attribute*, the `patch_id` has to be set for models with nested patches and internal coupling, otherwise `patch_id` is set to `current_patch`.

5.5.4.2 The subroutine `get_attribute`

`SUBROUTINE get_attribute`		`(status, (list, name) | (ganame) | (cname, caname) | (cname, oname, oaname) [,i] [,c] [,r] [,iflag] [,patch_id])`	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
list*)	TYPE(t_attribute_list)	POINTER	list of attributes
name*)	CHARACTER(LEN=*)	IN	name of attribute
ganame*)	CHARACTER(LEN=*)	IN	name of global attribute
cname*)	CHARACTER(LEN=*)	IN	name of channel
caname*)	CHARACTER(LEN=*)	IN	name of channel attribute
cname*)	CHARACTER(LEN=*)	IN	name of channel
oname*)	CHARACTER(LEN=*)	IN	name of object
oaname*)	CHARACTER(LEN=*)	IN	name of object attribute
optional arguments:			
i	INTEGER	OUT	integer value of attribute
c	CHARACTER(LEN=*)	OUT	string value of attribute
r	REAL(DP)	OUT	real value of attribute
iflag	INTEGER	OUT	check attribute at initialisation
patch_id*)	INTEGER	IN	number of patch

*)Note: This subroutine is fourfold overloaded to retrieve an *attribute* (`name`) from an *attribute list* (`list`), to retrieve a *global attribute* (`ganame`), to retrieve a *channel attribute* (`cname` is the *channel* name and `caname` the *channel attribute* name), or to retrieve a *channel object attribute* (`cname` is the *channel* name, `oname` the *channel object* name, `oaname` the *channel object attribute* name), respectively. `patch_id` is only available, when getting a *channel attribute* or *channel object attribute*.

This subroutine provides a comfortable interface to retrieve *attributes* of various entities, namely arbitrary *attribute lists* (`list` and `name`), centrally stored *global attributes* (`ganame`), *channel attributes* (`cname` and `caname`) and *channel object attributes* (`cname`, `oname` and `oaname`).

The type of the attribute is in all cases specified by parameter assignment to the corresponding optional argument for INTEGER (`i`), string (CHARACTER, `c`) or REAL(DP) (`r`), respectively. The argument `iflag` is the same as in `SUBROUTINE add_attribute` (Sect. 5.1.1).

When getting a *channel attribute* or *channel object attribute*, the `patch_id` has to be set for models with nested patches and internal coupling, otherwise `patch_id` is set to `current_patch`.

5.5.4.3 The subroutine `write_attribute`

SUBROUTINE write_attribute		(status [,cname | (cname, oname)] [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
cname*)	CHARACTER(LEN=*)	IN	name of channel
onamr*)	CHARACTER(LEN=*)	IN	name of object
optional arguments:			
patch_id*)	INTEGER	IN	number of patch

*)Note: This subroutine is threefold overloaded to write the *global attributes* (no second parameter), or to write the *channel attributes* (`cname` is the *channel* name) or the *channel object attributes* (`cname` is the *channel* name, `oname` the *channel object* name), respectively. `patch_id` is only available, when writing *channel attributes* or *channel object attributes*.

This subroutine writes all *global attributes* (no 2nd argument), the *attributes* of a specific *channel* (`cname`) or the attributes of a specific *channel object* (`cname, oname`) to the standard output. When writing *channel attributes* or *channel object attributes*, the `patch_id` has to be set for models with nested patches and internal coupling, otherwise `patch_id` is set to `current_patch`.

6 Channels and tracer

6.1 The file `messy_main_channel_tracer.f90`

The MESSy infrastructure submodel TRACER for the specific meta-data and memory management for constituents (such as water in different phases, chemical compounds, aerosol etc.) in different media and domains has been published by Jöckel et al. (2008)[5].

The subroutines described in this section comprise the interface between the TRACER memory management and the CHANNEL memory management, such that *tracers* (and their various instances, see reference above) are also *channel objects* and the complete namelist controlled output (see Sects. 2.1 and 2.2) can be applied.

6.1.1 The subroutine `create_tracer_channels`

SUBROUTINE create_tracer_channels		(status, trsetname, channelname, reprid)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
trsetname	CHARACTER(LEN=*)	IN	name of tracer set
channelname	CHARACTER(LEN=*)	IN	name of channel
reprid	INTEGER	IN	representation identifier

This subroutine creates a new *channel* (with name `channelname`) for the *tracers* in *tracer set* `trsetname`. In addition, depending on the construction of the tracer set, additional *channels* for all higher *tracer set instances* are created; those are named by the `channelname` suffixed with "_te" and "_m1" for the tendencies and $t - \Delta t$ values, respectively, or alternatively by "_nnn", where `nnn` is the number (with leading zeros) of the *tracer instance*.

All *instances* of all *tracers* in one *tracer set* share the same geometric layout, and therefore the same *representation*, which needs to be specified by its *representation* identifier (`reprid`).

Each *tracer* in a *tracer set* becomes associated with a *channel object* (with the tracer name) in all *channels* corresponding to the *instances* of the *tracer set*. The *tracer* memory and the *channel object* memory are identical through pointer association, no additional memory is required. The meta information of the *tracers* are internally converted to corresponding *channel object attributes*.

[5] http://www.atmos-chem-phys.net/8/1677

6.1.2 The subroutine `set_channel_or_tracer`

SUBROUTINE set_channel_or_tracer		(status ,trstr ,chstr ,cname ,oname ,pxt ,pxtte)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
trstr	CHARACTER(LEN=*)	IN	name of tracer set
chstr	CHARACTER(LEN=*)	IN	name of tracer channel
cname	CHARACTER(LEN=*)	IN	name of channel
oname	CHARACTER(LEN=*)	IN	name of object
pxt	REAL(DP), DIMENSION(:,:,:)	POINTER	pointer to (tracer) memory
pxtte	REAL(DP), DIMENSION(:,:,:)	POINTER	pointer to tracer tendency memory

This subroutine provides a comfortable interface to set pointers to the tracer memory (which is the same as the *channel object* memory), if the object (named `oname`) is a *tracer* in the *tracer set* `trstr`, or to the *channel object* memory, if the object is not a tracer, but an object in channel `cname`. In the first case, `pxt` points to the tracer memory and in addition, if the tracer set is constructed accordingly, `pxtte` points to the tracer tendency memory. In the second case, `pxt` points to the primary channel object memory, and `pxtte` is not associated.

7 Input/Output

7.1 The file `messy_main_channel_io.f90`

This module comprises the file format independent entry points for the output of *channels* and *channel objects* into output and restart files, and for the input of *channels* and *channel objects* from restart files. The subroutines of this module are called from within the basemodel interface layer (BMIL) and control the input / output of all *channels* and *channel objects* of all plugged-in submodels. This means, the subroutines are *collective* for all *channels* and *channel objects* in the model; nevertheless the output and restart file formats can be selected in the CTRL namelist for each channel individually (see Sect. 2.1).

The file format specific subroutines are implemented in the modules `messy_main_channel_netcdf.f90` and `messy_main_channel_pnetcdf.f90` for serial and parallel netCDF input / output (explained in Sects. 7.2 and 7.3), respectively.

7.1.1 The subroutine `initialize_parallel_io`

SUBROUTINE initialize_parallel_io		(status, p_pe, p_io, p_all_comm)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
p_pe	INTEGER	IN	parallel process identifier
p_io	INTEGER	IN	input / output process identifier
p_all_comm	INTEGER	IN	communicator

This subroutine is called once in the initialisation phase of the model, if parallel input / output is applied. The required information is stored internally, such as the current process identifier (`p_pe`), the process identifier for serial input / output (`p_io`), and the message passing interface (MPI) communicator (`p_all_comm`).

7.1.2 The subroutine `channel_init_restart`

SUBROUTINE channel_init_restart		(status, lp, lp_io, fname_base, rstatt)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
lp	LOGICAL	OUT	parallel input / output?
lp_io	LOGICAL	IN	input / output process?
fname_base	CHARACTER(LEN=*)	IN	basename of output file
rstatt	TYPE(t_attribute_list)	POINTER	list of restart attributes

This subroutine is called once during the initialisation phase of the model, if the model starts in restart mode. It internally sets the required information (e.g., timing information) read from one specific restart file, which name is `fname_base` suffixed by the file format extension, e.g., ".nc" for netCDF files. This is achieved by reading the *attributes* in the *attribute list* `rstatt` from the restart file. The parameter `lp` returns `.TRUE.`, if parallel input / output is used; the switch `lp_io` is used to limit diagnostic standard output to one process in a parallel environment.

7.1.3 The subroutine `channel_init_io`

SUBROUTINE channel_init_io		(status, lp_io, iomode, fname, amode [,att] [,chname] [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
lp_io	LOGICAL	IN	input / output process?
iomode	INTEGER	IN	read / write output or restart file?
fname	CHARACTER(LEN=*)	IN	name of output file
amode	INTEGER	IN	access mode (read or write)
optional arguments:			
att	TYPE(t_attribute_list)	POINTER	list of attributes
chname	CHARACTER(LEN=*)	IN	name of channel
patch_id	INTEGER	IN	number of patch

This subroutine is called once during the initialisation phase, if the model starts in restart mode, to initialise the input from the restart files, and / or once during the time integration phase of the model to initialise the output of data.

The mode is specified by the parameter `iomode`, which is `IOMODE_OUT` for output, or `IOMODE_RST` for restart files, respectively. The name of the output / restart file is composed by `fname`, followed (internally added) by an underscore, the name of the *channel* and the file format specific extension. The access mode `amode` is either `AMODE_WRITE` for output, or `AMODE_READ` for input (`IOMODE_RST` only!).

An optional list of *attributes* (`att`) can be specified for input / output.

The switch `lp_io` is used to limit diagnostic standard output to one process in a parallel environment.

For `iomode` = `AMODE_READ`, the optional parameter `chname` defines which *channel* to be read.

For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` = 1.

7.1.4 The subroutine `channel_write_header`

SUBROUTINE channel_write_header		(status ,lp_io ,iomode ,dimid_time [,att] [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
lp_io	LOGICAL	IN	input / output process?
iomode	INTEGER	IN	read / write output or restart file?
dimid_time	INTEGER	IN	time dimension identifier
optional arguments:			
att	TYPE(t_attribute_list)	POINTER	list of attributes
patch_id	INTEGER	IN	number of patch

This subroutine is called during the time integration phase of the model and outputs the header information for all *channels* to the corresponding output (`iomode=IOMODE_OUT`) or restart files (`iomode=IOMODE_RST`), respectively. For the output of time series, the *dimension* identifier of the time dimension `dimid_time` is required. Additional *attributes*, e.g., a special restart attribute list, can be added to the header by the optional parameter `att`.

The switch `lp_io` is used to limit diagnostic standard output to one process in a parallel environment.

For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` = 1.

7.1.5 The subroutine channel_write_time

SUBROUTINE channel_write_time		(status, lp_io, iomode, dimid_time [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
lp_io	LOGICAL	IN	input / output process?
iomode	INTEGER	IN	read / write output or restart file?
dimid_time	INTEGER	IN	time dimension identifier
optional arguments:			
patch_id	INTEGER	IN	number of patch

This subroutine is called during the time integration phase of the model and outputs the time information for all *channels* to the corresponding output (`iomode=IOMODE_OUT`) or restart files (`iomode=IOMODE_RST`), respectively. `dimid_time` is the time *dimension* identifier.

The switch `lp_io` is used to limit diagnostic standard output to one process in a parallel environment.

For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` = 1.

7.1.6 The subroutine channel_write_data

SUBROUTINE channel_write_data		(status, lp, lp_io, iomode, lexit, ptr, reprid [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
lp	LOGICAL	OUT	parallel input / output?
lp_io	LOGICAL	IN	input / output process?
iomode	INTEGER	IN	read / write output or restart file?
lexit	LOGICAL	OUT	loop finished?
ptr	REAL(DP), DIMENSION(:,:,:,:)	POINTER	data array for output
reprid	INTEGER	OUT	representation identifier
optional arguments:			
patch_id	INTEGER	IN	number of patch

This subroutine writes the *channel object* data arrays to the output (`iomode=IOMODE_OUT`) or restart files (`iomode=IOMODE_RST`), respectively.

It is called twice within an endless do-loop at the end of each model time step. This loop is complemented by an endless do-loop within the subroutine, which steps through all *channel object* data arrays (primary and secondary statistical). This nested loop construction is used to enable a separation of the parallel decomposition in the basemodel interface layer (BMIL).

The first call within the loop returns a pointer to the current *channel object* data array in parallel decomposition (memory layout, `ptr`) and the corresponding *representation* identifier (`reprid`) of the *channel object*. For serial output (`lp=.FALSE.`), `ptr` must then be recomposed (gathered) and – depending on the *representation* – reshaped into output layout. Alternatively, for parallel output (`lp=.TRUE.`), `ptr` must be – depending on the *representation* – only reshaped into output layout.

In the second call, `ptr` then represents the *channel object* data array in output layout, either decomposed (local) for parallel, or recomposed (global) for serial output.

If the last *channel object* data array has been passed in the inner loop, `lexit` is `.TRUE.` and the outer loop is exited.

The switch `lp_io` is used to limit diagnostic standard output to one process in a parallel environment.

For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` = 1.

7.1.7 The subroutine channel_finish_io

SUBROUTINE channel_finish_io		(status, lp_io, iomode, lclose [,chname] [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
lp_io	LOGICAL	IN	parallel input / output?
iomode	INTEGER	IN	read / write output or restart file?
lclose	LOGICAL	IN	close file after read / write?
optional arguments:			
chname	CHARACTER(LEN=*)	IN	name of channel
patch_id	INTEGER	IN	number of patch

This subroutine is called once after writing the data to finish the output to the output files (`iomode=IOMODE_OUT`) or restart files (`iomode=IOMODE_RST`), respectively. The files are either closed (`lclose=.TRUE.`), or remain open (`lclose=.FALSE.`), and the the corresponding output buffers are flushed (either system dependent, or forced by the namelist parameter `L_FLUSH_IOBUFFER`, see Sect. 2.1).

The switch `lp_io` is used to limit diagnostic standard output to one process in a parallel environment.

For `iomode` = `IOMODE_RST`, the optional parameter `chname` defines which *channel* to be read.

For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` = 1.

7.1.8 The subroutine channel_read_data

SUBROUTINE channel_read_data		(status, lp_io, iomode, lexit, ptr, reprid, lp [,chname] [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
lp_io	LOGICAL	IN	input / output process?
iomode	INTEGER	IN	read output or restart file?
lexit	LOGICAL	OUT	loop finished?
ptr	REAL(DP), DIMENSION(:,:,:,:)	POINTER	data array for input
reprid	INTEGER	OUT	representation identifier
lp	INTEGER	OUT	parallel input/ output?
optional arguments:			
chname	CHARACTER(LEN=*)	IN	name of channel
patch_id	INTEGER	IN	number of patch

This subroutine reads the *channel object* array data from the restart files (`iomode=IOMODE_RST`). It is called twice within an endless do-loop during the initialisation phase of the model, if it starts in restart mode. The loop is complemented by an endless do-loop within the subroutine, which steps through all *channel object* data arrays (primary and secondary statistical). This nested loop construction is used to enable a separation of the parallel decomposition in the basemodel interface layer (BMIL).

The first call within the loop returns a pointer to the current *channel object* data array in output layout (`ptr`), either decomposed for parallel (`lp=.TRUE.`), or recomposed for serial (`lp=.FALSE.`) input. The corresponding *representation* identifier of the *channel object* is stored in `reprid`. In the case of parallel input, `ptr` must then - depending on the *representation* - be reshaped to memory layout. Alternatively, for serial input, `ptr` must be decomposed (scattered) and - depending on the *representation* - reshaped to memory layout.

In the second call, `ptr` is represented in memory layout, either decomposed for parallel, or recomposed for serial input. The data is then copied to the corresponding *channel object* data array.

If the last *channel object* data array has been passed in the inner loop, `lexit` is `.TRUE.` and the outer loop is exited.

The switch `lp_io` is used to limit diagnostic standard output to one process in a parallel environment.

If `chanme` is specified, only data from the respective *channel* is read.

For models with nested patches and internal coupling, the `patch_id` has to be set, otherwise `patch_id` = 1.

7.2 The file messy_main_channel_netcdf.f90

This module contains the implementation for input / output from / into output and restart files in netCDF[6] format. All subroutines are called from within the corresponding subroutines in the module messy_main_channel_io.f90 (see Sect. 7.1).

7.2.1 The subroutine ch_netcdf_init_rst

SUBROUTINE ch_netcdf_init_rst		(status, fname, att)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
fname	CHARACTER(LEN=*)	IN	name of file
att	TYPE(t_attribute_list)	POINTER	list of attributes

This subroutine reads the restart *attributes* in the *attribute* list att from the file fname.

7.2.2 The subroutine ch_netcdf_init_io

SUBROUTINE ch_netcdf_init_io		(status, iomode, channel, amode [,att] [,patch_id])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
iomode	INTEGER	IN	read / write output or restart file?
channel	TYPE(t_channel)	POINTER	channel
amode	INTEGER	IN	access mode (read or write)
optional arguments:			
att	TYPE(t_attribute_list)	POINTER	attribute list
patch_id	INTEGER	IN	number of patch

This subroutine initialises the output (amode=AMODE_WRITE) of one channel into an output (iomode=IOMODE_OUT) or restart file (iomode=IOMODE_RST), respectively. Alternatively, the input (amode=AMODE_READ) of the channel from the corresponding restart file (iomode=IOMODE_RST) is initialised.

An optional list of *attributes* (att) can be specified for input / output.

For models with nested patches and internal coupling, the patch_id has to be set, otherwise patch_id = 1.

7.2.3 The subroutine ch_netcdf_write_header

SUBROUTINE ch_netcdf_write_header		(status ,iomode ,channel ,dim_time [,att])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
iomode	INTEGER	IN	write output or restart file?
channel	TYPE(t_channel)	POINTER	channel
dim_time	TYPE(t_dimension)	POINTER	time dimension
optional arguments:			
att	TYPE(t_attribute_list)	POINTER	list of attributes

This subroutine writes the header information of one channel to the corresponding output (iomode=IOMODE_OUT) or restart file (iomode=IOMODE_RST), respectively. For the time series output, the time *dimension* dim_time is required. Additional *attributes* (e.g., the restart attributes) can optionally be specified by the *attribute* list att.

[6] http://www.unidata.ucar.edu/software/netcdf

7.2.4 The subroutine ch_netcdf_write_time

SUBROUTINE ch_netcdf_write_time		(status, channel, dim_time)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
channel	TYPE(t_channel)	POINTER	channel
dim_time	TYPE(t_dimension)	POINTER	time dimension

This subroutine writes the time information, specified by the time *dimension* `dim_time` to the output file of the `channel`.

7.2.5 The subroutine ch_netcdf_write_data

SUBROUTINE ch_netcdf_write_data		(status, iomode, channel, object, ptr, jsnd, i2nd)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
iomode	INTEGER	IN	write output or restart file?
channel	TYPE(t_channel)	POINTER	channel
object	TYPE(t_channel_object)	POINTER	object
ptr	REAL(DP), DIMENSION(:,:,:,:)	POINTER	data array
jsnd	INTEGER	IN	output data type
i2nd	INTEGER	IN	index of secondary array

This subroutine writes one *channel object* data array of the `object` in `channel` to the corresponding output (`iomode=IOMODE_OUT`) or restart file (`iomode=IOMODE_RST`), respectively. `ptr` is the pointer to the actual data array in memory, `jsnd` indicates the type of primary or secondary (derived statistical) data (i.e., `SND_INS`, `SND_AVE`, `SND_STP`, `SND_MIN`, `SND_MAX`, `SND_CNT`, `SND_CAV`), and `i2nd` is the position of the actual secondary data in the internal secondary data array.

7.2.6 The subroutine ch_netcdf_finish_io

SUBROUTINE ch_netcdf_finish_io		(status, iomode, channel, lclose)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
iomode	INTEGER	IN	write output or restart file?
channel	TYPE(t_channel)	POINTER	channel
lclose	LOGICAL	IN	close file after read / write?

This subroutine closes (`lclose=.TRUE.`) or flushes (`lclose=.FALSE.`) the output (`iomode=IOMODE_OUT`) or restart file (`iomode=IOMODE_RST`) of `channel`, respectively. The flushing is either system dependent (output buffer size), or can be forced by the namelist parameter `L_FLUSH_IOBUFFER` (see Sect. 2.1).

7.2.7 The subroutine ch_netcdf_read_data

SUBROUTINE ch_netcdf_read_data		(status, iomode, channel, object, ptr, jsnd, i2nd)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
iomode	INTEGER	IN	read output or restart file?
channel	TYPE(t_channel)	POINTER	channel
object	TYPE(t_channel_object)	POINTER	object
ptr	REAL(DP), DIMENSION(:,:,:,:)	POINTER	data array
jsnd	INTEGER	IN	output data type
i2nd	INTEGER	IN	index of secondary array

This subroutine reads one *channel object* data array of the `object` in `channel` from the corresponding restart file (`iomode=IOMODE_RST`). `ptr` is the pointer to the data in output layout. `jsnd` indicates the type of primary or secondary (derived statistical) data (i.e., `SND_INS`, `SND_AVE`, `SND_STP`, `SND_MIN`, `SND_MAX`, `SND_CNT`, `SND_CAV`), and `i2nd` is the position of the actual secondary data in the internal secondary data array.

7.3 The file `messy_main_channel_pnetcdf.f90`

This module contains the implementation for input / output from / into output and restart files in netCDF format by using the parallel netCDF[7] library. All subroutines are called from within the corresponding subroutines in the module `messy_main_channel_io.f90` (see Sect. 7.1).

7.3.1 The subroutine `ch_pnetcdf_init_pio`

SUBROUTINE ch_pnetcdf_init_pio		(status ,ex_p_pe ,ex_p_io ,ex_p_all_comm)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
ex_p_pe	INTEGER	IN	parallel process identifier
ex_p_io	INTEGER	IN	input / output process identifier
ex_p_all_comm	INTEGER	IN	communicator

With this subroutine the settings required for parallel input / output are stored internally. These are the current process identifier (`ex_p_pe`), the process identifier for serial input / output (`ex_p_io`), and the message passing interface (MPI) communicator (`ex_p_all_comm`).

7.3.2 The subroutine `ch_pnetcdf_init_rst`

SUBROUTINE ch_netcdf_init_rst		(status, fname, att)	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
fname	CHARACTER(LEN=*)	IN	name of file
att	TYPE(t_attribute_list)	POINTER	list of attributes

This subroutine reads the restart *attributes* in the *attribute* list `att` from the file `fname`.

7.3.3 The subroutine `ch_pnetcdf_init_io`

SUBROUTINE ch_pnetcdf_init_io		(status, iomode, channel, amode [,att])	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
iomode	INTEGER	IN	read / write output or restart file?
channel	TYPE(t_channel)	POINTER	channel
amode	INTEGER	IN	access mode (read or write)
optional arguments:			
att	TYPE(t_attribute_list)	POINTER	list of attributes

This subroutine initialises the output (`amode=AMODE_WRITE`) of one `channel` into an output (`iomode=IOMODE_OUT`) or restart file (`iomode=IOMODE_RST`), respectively. Alternatively, the input (`amode=AMODE_READ`) of the `channel` from the corresponding restart file (`iomode=IOMODE_RST`) is initialised.

An optional list of *attributes* (`att`) can be specified for input / output.

[7] `http://www.mcs.anl.gov/parallel-netcdf`

7.3.4 The subroutine `ch_pnetcdf_write_header`

`SUBROUTINE ch_pnetcdf_write_header`	`(status, iomode, channel, dim_time [,att])`		
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
iomode	INTEGER	IN	write output or restart file?
channel	TYPE(t_channel)	POINTER	channel
dim_time	TYPE(t_dimension)	POINTER	time dimension
optional arguments:			
att	TYPE(t_attribute_list)	POINTER	list of attributes

This subroutine writes the header information of one `channel` to the corresponding output (`iomode=IOMODE_OUT`) or restart file (`iomode=IOMODE_RST`), respectively. For the time series output, the time *dimension* `dim_time` is required. Additional *attributes* (e.g., the restart attributes) can optionally be specified by the *attribute* list `att`.

7.3.5 The subroutine `ch_pnetcdf_write_time`

`SUBROUTINE ch_pnetcdf_write_time`	`(status, channel, dim_time)`		
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
channel	TYPE(t_channel)	POINTER	channel
dim_time	TYPE(t_dimension)	POINTER	time dimension

This subroutine writes the time information, specified by the time *dimension* `dim_time` to the output file of the `channel`.

7.3.6 The subroutine `ch_pnetcdf_write_data`

`SUBROUTINE ch_pnetcdf_write_data`	`(status ,iomode ,channel ,object ,ptr ,jsnd ,i2nd)`		
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
iomode	INTEGER	IN	write output or restart file?
channel	TYPE(t_channel)	POINTER	channel
object	TYPE(t_channel_object)	POINTER	object
ptr	REAL(DP), DIMENSION(:,:,:,:)	POINTER	data array
jsnd	INTEGER	IN	output data type
i2nd	INTEGER	IN	index of secondary array

This subroutine writes one *channel object* data array of the `object` in `channel` to the corresponding output (`iomode=IOMODE_OUT`) or restart file (`iomode=IOMODE_RST`), respectively. `ptr` is the pointer to the actual data array in memory, `jsnd` indicates the type of primary or secondary (derived statistical) data (i.e., `SND_INS`, `SND_AVE`, `SND_STP`, `SND_MIN`, `SND_MAX`, `SND_CNT`, `SND_CAV`), and `i2nd` is the position of the actual secondary data in the internal secondary data array.

7.3.7 The subroutine `ch_pnetcdf_finish_io`

`SUBROUTINE ch_pnetcdf_finish_io`	`(status, iomode, channel, lclose)`		
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
iomode	INTEGER	IN	read / write output or restart file?
channel	TYPE(t_channel)	POINTER	channel
lclose	LOGICAL	IN	close file after read / write?

This subroutine closes (`lclose=.TRUE.`) or flushes (`lclose=.FALSE.`) the output (`iomode=IOMODE_OUT`) or restart file (`iomode=IOMODE_RST`) of `channel`, respectively. The flushing is either system dependent (output buffer size), or can be forced by the `L_FLUSH_IOBUFFER` namelist parameter (see Sect. 2.1).

7.3.8 The subroutine `ch_pnetcdf_read_data`

`SUBROUTINE ch_pnetcdf_read_data`		`(status, iomode, channel, object, ptr, jsnd, i2nd)`	
name	type	intent	description
mandatory arguments:			
status	INTEGER	OUT	
iomode	INTEGER	IN	read output or restart file?
channel	TYPE(t_channel)	POINTER	channel
object	TYPE(t_channel_object)	POINTER	object
ptr	REAL(DP), DIMENSION(:,:,:,:)	POINTER	data array
jsnd	INTEGER	IN	output data type
i2nd	INTEGER	IN	index of secondary array

This subroutine reads one *channel object* data array of the `object` in `channel` from the corresponding restart file (`iomode=IOMODE_RST`). `ptr` is the pointer to the data in output layout. `jsnd` indicates the type of primary or secondary (derived statistical) data (i.e., `SND_INS`, `SND_AVE`, `SND_STP`, `SND_MIN`, `SND_MAX`, `SND_CNT`, `SND_CAV`), and `i2nd` is the position of the actual secondary data in the internal secondary data array.

7.4 The implementation of alternative input / output formats

The file format specific subroutines (e.g., for netCDF, Sect. 7.2 and parallel netCDF, Sect. 7.3) are all called from within the corresponding subroutines in the main input / output module `messy_main_channel_io.f90` (Sect. 7.1). This implies that a basemodel that is already equipped with the CHANNEL submodel can easily be extended by additional output formats and / or methods, since only CHANNEL needs to be extended. For such an extension only a few steps are required:

First, a new module `messy_main_channel_`*format*`.f90` needs to be written, where *format* specifies the new file format. This new module must contain all subroutines equivalent to the corresponding subroutines in `messy_main_channel_netcdf.f90` (Sect. 7.2) or `messy_main_channel_pnetcdf.f90` (Sect. 7.3).

Second, the new subroutines from `messy_main_channel_`*format*`.f90` must be called from within the CASE-constructs in `messy_main_channel_io.f90`. The latter provides all required main entry points for input / output; exemplarily the entry points for ASCII, GRIB, HDF4 and HDF5 formats are already prepared.

As a third step, it might be required to modify (extend) the module `messy_main_channel.f90` at the positions marked with

```
! +++ ADD OTHER OUTPUT FORMATS HERE
```

These modifications comprise

- a *channel* specific structure definition to store file format related meta information, e.g., the logical input / output unit etc. (equivalent to `t_channel_object_netcdf`, see Sect. 3.5),
- a *channel object* specific structure definition to store file format related meta information, e.g., variable identifiers etc. (equivalent to `t_channel_netcdf`, see Sect. 3.6),
- potentially two short code sequences to manage the additional meta information for the secondary data arrays.

The information stored in these additional structures are then applied in the respective subroutines of `messy_main_channel_`*format*`.f90`.

8 A documented example

This section contains the README of the example code, which is part of the CHANNEL distribution.

```
================================================================================
THIS README CONTAINS A BRIEF DESCRIPTION OF THE MESSy GENERIC SUBMODEL CHANNEL
FILES AND THE SIMPLIFIED BASEMODEL/SUBMODEL.

A HIGHLY SIMPLIFIED BASE MODEL LAYER (BML)
==========================================
(.) channel_bml.f90                  ! initialisation, time loop, finish
(.) channel_bml_mem.f90              ! declaraion of parameters

AN EXEMPLAYRY BASEMODEL INTERFACE LAYER (BMIL) FOR THE SIMPLIFIED BASEMODEL
============================================================================
(+) messy_main_channel_bi.f90        ! create the basemodel specific channel
                                     ! setup

The basemodel (with name CHANNEL)
- initialises an exemplary channel environment with 4 representations:
  - GP_3D_MID with rank 3 (longitude x latitude x level) and dimensions
                           36        x 18       x 2
  - GP_3D_MID_BND with rank 3 (longitude x level x latitude) and dimensions
                           36        x 2       x 18
                  and 2 boundary boxes in longitude and latitude direction
  - SCALAR    with rank 0
  - ARRAY     with rank 1 and the dimension of the intrinsic fortran random
              number generator state vector
- defines a standard stream (CHANNEL) with some SCALAR objects that contain the
  current system time
- detects if it is started 'from scratch', or if it is continued (restart);
  the latter is triggered by the presence of a file restart_NNNN_CHANNEL.nc,
  where NNNN is the restart cycle number (between 0001 and 9999); the restart
  file with the largest cycle number is used for continuation.
- initialises the simplified submodel (see below)
- contains a time loop in which it
  - calls the submodel to create some artifical (random) data (see below)
  - writes output and/or restart information to netCDF files
- cleans up the memory

THE GENERIC SUBMODEL CHANNEL
============================
SUBMODEL CORE LAYER (SMCL):
(*)  messy_main_constants_mem.f90    ! SMCL constants for various MESSy SMs
                                     ! (Not all of the contents is needed in
                                     !  the simplified CHANNEL box model.)
(*)  messy_main_tools.f90            ! SMCL tools for varius MESSy submodels
                                     ! (Not all of the contents is needed in
                                     !  the simplified CHANNLE box model.)
(*)  messy_main_blather.f90          ! SMCL log-message tools for various MESSy
                                     ! submodels. (Not all of the contents is
     ! needed in the simplified CHANNLE box
                                     ! model.)

(*)  messy_main_channel_error.f90      ! error numbers and messages
(*)  messy_main_channel_attributes.f90 ! types and routines for ... attributes,
(*)  messy_main_channel_dimensions.f90 ! ... dimensions,
```

```
(*)  messy_main_channel_dimvar.f90     ! ... dimension-variables,
(*)  messy_main_channel_repr.f90       ! ... representations,
(*)  messy_main_channel.f90            ! ... channels and channel-objects
(*)  messy_main_channel_tracer.f90-bak ! association of tracer memory from
                                       ! MESSy generic submodel TRACER to
                                       ! channel memory; Note: this is not
                                       ! used in this simplified example.

(*)  messy_main_channel_io.f90         ! main module for handling I/O
(*)  messy_main_channel_netcdf.f90     ! I/O in netCDF format
(*)  messy_main_channel_pnetcdf.f90    ! I/O based on parallel-netCDF
                                       ! Note: The simplified example has no
                                       ! parallel mode!

USER INTERFACE (UI):
(*)  channel.nml                      ! user interface (namelist file)
                                      ! with CTRL and CPL namelists

AN EXEMPLARY ORDINARY SUBMODEL
==============================
(-)  messy_submodel.f90
(-)  messy_submodel_si.f90

This simple submodel defines a new channel and four objects in different
representations:
 - f01: a 3-d field in GP_3D_MID representation (longitude x latitude x level)
 - fbnd: a 3-d field in GP_3D_MID_BND representation
   (longitude x level x latitude) with additional 2 boundary boxes,
   both, in longitud and latitude direction, which are accessed internally,
   but do not appear in the output
 - s01: a SCALAR
 - state: an ARRAY
f01 is filled with random numbers, s01 is the avarage of f01. In state, the
state vector of the fortran random number generator is saved to be able to
continue a pseudo-random number series after a basemodel restart.
Note: The ARRAY-length is the length of the intrinisic random number state
vector and therefore depends on the Fortran system.

NOTES:
======
(*)  These files/modules need NOT to be changed in order to use CHANNEL in
     another basemodel.
(+)  These files/modules need to be rewritten for any other basemodel. The
     examples of the simple basemodel here, however, show the overall structure
     and the usage of the CHANNEL interface.
(.)  These files/modules constitute a highly simplified basemodel and are
     usually replaced by a specific model.
(-)  These files/modules represent a simple MESSy submodel which is useless
     outside the context of the simpified basemodel.

HOWTO COMPILE/RUN
=================
1. Edit the file 'Makefile' and put in your specific compiler settings and
   the path to the netCDF library.
   Do not forget to activate the pre-processor in your compiler options!

2. > gmake
```

```
   Note: The 'Makefile' includes 'main.mk' and 'depend.mk'. The latter
         contains the file dependencies, which are automatically updated
         (see 5 below).
   Note: Other usefule gmake-targets are 'gmake clean' and 'gmake distclean'.

3. > gmake run

4. Change entries in channel.nml and go back to 3.

5. Change the code (e.g. channel_bml_mem.f90) and go back to 2,3,4.
   Note: The file dependencies are automatically updated with the perl-script
         'sfmakedepend'. Make sure that the path to 'perl' in the very first
         line of this script is OK.
================================================================================
```

References

Jöckel, P., Sander, R., Kerkweg, A., Tost, H., and Lelieveld, J.: Technical Note: The Modular Earth Submodel System (MESSy) - a new approach towards Earth System Modeling, Atmos. Chem. Phys., 5, 433–444, http://www.atmos-chem-phys.net/8/1677, 2005.

Jöckel, P., Kerkweg, A., Buchholz-Dietsch, J., Tost, H., Sander, R., and Pozzer, A.: Technical Note: Coupling of chemical processes with the Modular Earth Submodel System (MESSy) submodel TRACER, Atmos. Chem. Phys., 8, 1677–1687, http://www.atmos-chem-phys.net/8/1677, 2008.