CDI Fortran Manual

Climate Data Interface
Version 1.7.0
October 2015

Uwe Schulzweida
Max-Planck-Institute for Meteorology

Contents

1. Introduction

CDI is an Interface to access Climate and forecast model Data. The interface is independent from a specific data format and has a C and Fortran API. **CDI** was developed for a fast and machine independent access to GRIB and netCDF datasets with the same interface. The local MPI-MET data formats SERVICE, EXTRA and IEG are also supported.

1.1. Building from sources

This section describes how to build the **CDI** library from the sources on a UNIX system. **CDI** is using the GNU configure and build system to compile the source code. The only requirement is a working ANSI C99 compiler.
First go to the `download` page (`http://code.zmaw.de/projects/cdi/files`) to get the latest distribution, if you do not already have it.
To take full advantage of **CDI**'s features the following additional libraries should be installed:

- Unidata netCDF library (`http://www.unidata.ucar.edu/packages/netcdf`) version 3 or higher. This is needed to read/write netCDF files with **CDI**.
- The ECMWF GRIB_API (`http://www.ecmwf.int/products/data/software/grib_api.html`) version 1.9.5 or higher. This library is needed to encode/decode GRIB2 records with **CDI**.

1.1.1. Compilation

Compilation is now done by performing the following steps:

1. Unpack the archive, if you haven't already done that:

```
gunzip cdi-$VERSION.tar.gz    # uncompress the archive
tar xf cdi-$VERSION.tar       # unpack it
cd cdi-$VERSION
```

2. Run the configure script:

```
./configure
```

Or optionally with netCDF support:

```
./configure --with-netcdf=<netCDF root directory>
```

For an overview of other configuration options use

```
./configure --help
```

3. Compile the program by running make:

```
make
```

The software should compile without problems and the **CDI** library (`libcdi.a`) should be available in the `src` directory of the distribution.

1.1.2. Installation

After the compilation of the source code do a `make install`, possibly as root if the destination permissions require that.

```
make install
```

The library is installed into the directory `<prefix>/lib`. The C and Fortran include files are installed into the directory `<prefix>/include`. `<prefix>` defaults to `/usr/local` but can be changed with the `--prefix` option of the configure script.

2. File Formats

2.1. GRIB

GRIB [GRIB] (GRIdded Binary) is a standard format designed by the World Meteorological Organization (WMO) to support the efficient transmission and storage of gridded meteorological data.

A GRIB record consists of a series of header sections, followed by a bitstream of packed data representing one horizontal grid of data values. The header sections are intended to fully describe the data included in the bitstream, specifying information such as the parameter, units, and precision of the data, the grid system and level type on which the data is provided, and the date and time for which the data are valid.

Non-numeric descriptors are enumerated in tables, such that a 1-byte code in a header section refers to a unique description. The WMO provides a standard set of enumerated parameter names and level types, but the standard also allows for the definition of locally used parameters and geometries. Any activity that generates and distributes GRIB records must also make their locally defined GRIB tables available to users.

CDI does not support the full GRIB standard. The following data representation and level types are implemented:

GRIB1 grid type	GRIB2 template	GRIB_API name	description
0	3.0	regular_ll	Regular longitude/latitude grid
3	–	lambert	Lambert conformal grid
4	3.40	regular_gg	Regular Gaussian longitude/latitude grid
4	3.40	reduced_gg	Reduced Gaussian longitude/latitude grid
10	3.1	rotated_ll	Rotated longitude/latitude grid
50	3.50	sh	Spherical harmonic coefficients
192	3.100	–	Icosahedral-hexagonal GME grid
–	3.101	–	General unstructured grid

GRIB1 level type	GRIB2 level type	GRIB_API name	description
1	1	surface	Surface level
2	2	cloudBase	Cloud base level
3	3	cloudTop	Level of cloud tops
4	4	isothermZero	Level of 0° C isotherm
8	8	nominalTop	Norminal top of atmosphere
9	9	seaBottom	Sea bottom
10	10	entireAtmosphere	Entire atmosphere
99	–	–	Isobaric level in Pa
100	100	isobaricInhPa	Isobaric level in hPa
102	101	meanSea	Mean sea level
103	102	heightAboveSea	Altitude above mean sea level
105	103	heightAboveGround	Height level above ground
107	104	sigma	Sigma level
109	105	hybrid	Hybrid level
110	105	hybridLayer	Layer between two hybrid levels
111	106	depthBelowLand	Depth below land surface
112	106	depthBelowLandLayer	Layer between two depths below land surface
113	107	theta	Isentropic (theta) level
–	114	–	Snow level
160	160	depthBelowSea	Depth below sea level
162	162	–	Lake or River Bottom
163	163	–	Bottom Of Sediment Layer
164	164	–	Bottom Of Thermally Active Sediment Layer
165	165	–	Bottom Of Sediment Layer Penetrated By Thermal Wave
166	166	–	Mixing Layer

2.1.1. GRIB edition 1

GRIB1 is implemented in **CDI** as an internal library and enabled per default. The internal GRIB1 library is called CGRIBEX. This is lightweight version of the ECMWF GRIBEX library. CGRIBEX is written in ANSI C with a portable Fortran interface. The configure option `--disable-cgribex` will disable the encoding/decoding of GRIB1 records with CGRIBEX.

2.1.2. GRIB edition 2

GRIB2 is available in **CDI** via the ECMWF GRIB_API [GRIBAPI]. GRIB_API is an external library and not part of **CDI**. To use GRIB2 with **CDI** the GRIB_API library must be installed before the configuration of the **CDI** library. Use the configure option `--with-grib_api` to enable GRIB2 support.
The GRIB_API library is also used to encode/decode GRIB1 records if the support for the CGRIBEX library is disabled. This feature is not tested regulary and the status is experimental!

2.2. NetCDF

NetCDF [NetCDF] (Network Common Data Form) is an interface for array-oriented data access and a library that provides an implementation of the interface. The netCDF library also defines a machine-independent format for representing scientific data. Together, the interface, library, and format support the creation, access, and sharing of scientific data.

CDI only supports the classic data model of netCDF and arrays up to 4 dimensions. These dimensions should only be used by the horizontal and vertical grid and the time. The netCDF attributes should follow the GDT, COARDS or CF Conventions.
NetCDF is an external library and not part of **CDI**. To use netCDF with **CDI** the netCDF library must be installed before the configuration of the **CDI** library. Use the configure option `--with-netcdf` to enable netCDF support (see Build).

2.3. SERVICE

SERVICE is the binary exchange format of the atmospheric general circulation model ECHAM [ECHAM]. It has a header section with 8 integer values followed by the data section. The header and the data section have the standard Fortran blocking for binary data records. A SERVICE record can have an accuracy of 4 or 8 bytes and the byteorder can be little or big endian. In **CDI** the accuracy of the header and data section must be the same. The following Fortran code example can be used to read a SERVICE record with an accuracy of 4 bytes:

```
INTEGER*4 icode,ilevel,idate,itime,nlon,nlat,idispo1,idispo2
REAL*4 field(mlon,mlat)
  ...
READ(unit) icode,ilevel,idate,itime,nlon,nlat,idispo1,idispo2
READ(unit) ((field(ilon,ilat), ilon=1,nlon), ilat=1,nlat)
```

The constants `mlon` and `mlat` must be greater or equal than `nlon` and `nlat`. The meaning of the variables are:

`icode`	The code number
`ilevel`	The level
`idate`	The date as YYYYMMDD
`itime`	The time as hhmmss
`nlon`	The number of longitudes
`nlat`	The number of latitides
`idispo1`	For the users disposal (Not used in **CDI**)
`idispo2`	For the users disposal (Not used in **CDI**)

SERVICE is implemented in **CDI** as an internal library and enabled per default. The configure option `--disable-service` will disable the support for the SERVICE format.

2.4. EXTRA

EXTRA is the standard binary output format of the ocean model MPIOM [MPIOM]. It has a header section with 4 integer values followed by the data section. The header and the data section have the standard Fortran blocking for binary data records. An EXTRA record can have an accuracy of 4 or 8 bytes and the byteorder can be little or big endian. In **CDI** the accuracy of the header and data section must be the same. The following Fortran code example can be used to read an EXTRA record with an accuracy of 4 bytes:

```
INTEGER*4 idate,icode,ilevel,nsize
REAL*4 field(msize)
  ...
READ(unit) idate,icode,ilevel,nsize
READ(unit) (field(isize),isize=1,nsize)
```

The constant `msize` must be greater or equal than `nsize`. The meaning of the variables are:

`idate`	The date as YYYYMMDD
`icode`	The code number
`ilevel`	The level
`nsize`	The size of the field

EXTRA is implemented in **CDI** as an internal library and enabled per default. The configure option `--disable-extra` will disable the support for the EXTRA format.

2.5. IEG

IEG is the standard binary output format of the regional model REMO [REMO]. It is simple an unpacked GRIB edition 1 format. The product and grid description sections are coded with 4 byte integer values and the data section can have 4 or 8 byte IEEE floating point values. The header and the data section have the standard Fortran blocking for binary data records. The IEG format has a fixed size of 100 for the vertical coordinate table. That means it is not possible to store more than 50 model levels with this format. **CDI** supports only data on Gaussian and LonLat grids for the IEG format.

IEG is implemented in **CDI** as an internal library and enabled per default. The configure option `--disable-ieg` will disable the support for the IEG format.

3. Use of the CDI Library

This chapter provides templates of common sequences of **CDI** calls needed for common uses. For clarity only the names of routines are used. Declarations and error checking were omitted. Statements that are typically invoked multiple times were indented and ... is used to represent arbitrary sequences of other statements. Full parameter lists are described in later chapters. Complete examples for write, read and copy a dataset with **CDI** can be found in Appendix B.

3.1. Creating a dataset

Here is a typical sequence of **CDI** calls used to create a new dataset:

```
    gridCreate           ! create a horizontal Grid: from type and size
       ...
    zaxisCreate          ! create a vertical Z-axis: from type and size
       ...
    taxisCreate          ! create a Time axis: from type
       ...
    vlistCreate          ! create a variable list
          ...
       vlistDefVar       ! define variables: from Grid and Z-axis
          ...
    streamOpenWrite      ! create a dataset: from name and file type
       ...
    streamDefVlist       ! define variable list
       ...
    streamDefTimestep    ! define time step
          ...
       streamWriteVar    ! write variable
          ...
    streamClose          ! close the dataset
       ...
    vlistDestroy         ! destroy the variable list
       ...
    taxisDestroy         ! destroy the Time axis
       ...
    zaxisDestroy         ! destroy the Z-axis
       ...
    gridDestroy          ! destroy the Grid
```

3.2. Reading a dataset

Here is a typical sequence of **CDI** calls used to read a dataset:

```
    streamOpenRead       ! open existing dataset
       ...
    streamInqVlist       ! find out what is in it
          ...
       vlistInqVarGrid   ! get an identifier to the Grid
          ...
```

```
      vlistInqVarZaxis   ! get an identifier to the Z-axis
         ...
      vlistInqTaxis      ! get an identifier to the T-axis
         ...
   streamInqTimestep     ! get time step
         ...
      streamReadVar      ! read varible
         ...
   streamClose           ! close the dataset
```

3.3. Compiling and Linking with the CDI library

Details of how to compile and link a program that uses the **CDI** C or FORTRAN interfaces differ, depending on the operating system, the available compilers, and where the **CDI** library and include files are installed. Here are examples of how to compile and link a program that uses the **CDI** library on a Unix platform, so that you can adjust these examples to fit your installation. There are two different interfaces for using **CDI** functions in Fortran: `cfortran.h` and the instrinsic `iso_c_binding` module from Fortran 2003 standard. At first, the preparations for compilers without F2003 capabilities are described.

Every FORTRAN file that references **CDI** functions or constants must contain an appropriate `INCLUDE` statement before the first such reference:

```
INCLUDE "cdi.inc"
```

Unless the `cdi.inc` file is installed in a standard directory where FORTRAN compiler always looks, you must use the `-I` option when invoking the compiler, to specify a directory where `cdi.inc` is installed, for example:

```
f77 -c -I/usr/local/cdi/include myprogram.f
```

Alternatively, you could specify an absolute path name in the `INCLUDE` statement, but then your program would not compile on another platform where **CDI** is installed in a different location. Unless the **CDI** library is installed in a standard directory where the linker always looks, you must use the `-L` and `-l` options to links an object file that uses the **CDI** library. For example:

```
f77 -o myprogram myprogram.o -L/usr/local/cdi/lib -lcdi
```

Alternatively, you could specify an absolute path name for the library:

```
f77 -o myprogram myprogram.o -L/usr/local/cdi/lib/libcdi
```

If the **CDI** library is using other external libraries, you must add this libraries in the same way. For example with the netCDF library:

```
f77 -o myprogram myprogram.o -L/usr/local/cdi/lib -lcdi \
                             -L/usr/local/netcdf/lib -lnetcdf
```

For using the `iso_c_bindings` two things are necessary in a program or module

```
USE ISO_C_BINDING
USE mo_cdi
```

The iso_c_binding module is included in mo_cdi, but without cfortran.h characters and character variables have to be handled separately. Examples are available in section B.4.
After installation mo_cdi.o and mo_cdi.mod are located in the library and header directory respectively. cdilib.o has to be mentioned directly on the command line. It can be found in the library directory, too. Depending on the **CDI** configuration, a compile command should look like this:

```
nagf95 -f2003 -g cdi_read_f2003.f90 -L/usr/lib -lnetcdf -o cdi_read_example
                                    -I/usr/local/include
                                    /usr/local/lib/cdilib.o /usr/local/lib/mo_cdi.o
```

4. CDI modules

4.1. Dataset functions

This module contains functions to read and write the data. To create a new dataset the output format must be specified with one of the following predefined file format types:

FILETYPE_GRB	File type GRIB version 1
FILETYPE_GRB2	File type GRIB version 2
FILETYPE_NC	File type netCDF
FILETYPE_NC2	File type netCDF version 2 (64-bit)
FILETYPE_NC4	File type netCDF-4 (HDF5)
FILETYPE_NC4C	File type netCDF-4 classic
FILETYPE_SRV	File type SERVICE
FILETYPE_EXT	File type EXTRA
FILETYPE_IEG	File type IEG

`FILETYPE_GRB2` is only available if the **CDI** library was compiled with GRIB_API support and all netCDF file types are only available if the **CDI** library was compiled with netCDF support!
To set the byte order of a binary dataset with the file format type `FILETYPE_SRV`, `FILETYPE_EXT` or `FILETYPE_IEG` use one of the following predefined constants in the call to `streamDefByteorder`:

CDI_BIGENDIAN	Byte order big endian
CDI_LITTLEENDIAN	Byte order little endian

4.1.1. Create a new dataset: `streamOpenWrite`

The function `streamOpenWrite` creates a new datset.

Usage

```
INTEGER FUNCTION streamOpenWrite(CHARACTER*(*) path, INTEGER filetype)
```

`path` The name of the new dataset.

`filetype` The type of the file format, one of the set of predefined **CDI** file format types. The valid **CDI** file format types are `FILETYPE_GRB`, `FILETYPE_GRB2`, `FILETYPE_NC`, `FILETYPE_NC2`, `FILETYPE_NC4`, `FILETYPE_NC4C`, `FILETYPE_SRV`, `FILETYPE_EXT` and `FILETYPE_IEG`.

Result

Upon successful completion `streamOpenWrite` returns an identifier to the open stream. Otherwise, a negative number with the error status is returned.

Errors

`CDI_ESYSTEM`	Operating system error.
`CDI_EINVAL`	Invalid argument.
`CDI_EUFILETYPE`	Unsupported file type.
`CDI_ELIBNAVAIL`	Library support not compiled in.

Example

Here is an example using `streamOpenWrite` to create a new netCDF file named `foo.nc` for writing:

```
INCLUDE 'cdi.h'
   ...
INTEGER streamID
   ...
streamID = streamOpenWrite("foo.nc", FILETYPE_NC)
IF ( streamID .LT. 0 ) CALL handle_error(streamID)
   ...
```

4.1.2. Open a dataset for reading: `streamOpenRead`

The function `streamOpenRead` opens an existing dataset for reading.

Usage

```
INTEGER FUNCTION streamOpenRead(CHARACTER*(*) path)
```

`path`	The name of the dataset to be read.

Result

Upon successful completion `streamOpenRead` returns an identifier to the open stream. Otherwise, a negative number with the error status is returned.

Errors

`CDI_ESYSTEM`	Operating system error.
`CDI_EINVAL`	Invalid argument.
`CDI_EUFILETYPE`	Unsupported file type.
`CDI_ELIBNAVAIL`	Library support not compiled in.

Example

Here is an example using `streamOpenRead` to open an existing netCDF file named `foo.nc` for reading:

```
INCLUDE 'cdi.h'
   ...
INTEGER streamID
   ...
streamID = streamOpenRead("foo.nc")
IF ( streamID .LT. 0 ) CALL handle_error(streamID)
   ...
```

4.1.3. Close an open dataset: `streamClose`

The function `streamClose` closes an open dataset.

Usage

```
SUBROUTINE streamClose(INTEGER streamID)
```

`streamID` Stream ID, from a previous call to `streamOpenRead` or `streamOpenWrite`.

4.1.4. Get the filetype: `streamInqFiletype`

The function `streamInqFiletype` returns the filetype of a stream.

Usage

```
INTEGER FUNCTION streamInqFiletype(INTEGER streamID)
```

`streamID` Stream ID, from a previous call to `streamOpenRead` or `streamOpenWrite`.

Result

`streamInqFiletype` returns the type of the file format, one of the set of predefined **CDI** file format types. The valid **CDI** file format types are `FILETYPE_GRB`, `FILETYPE_GRB2`, `FILETYPE_NC`, `FILETYPE_NC2`, `FILETYPE_NC4`, `FILETYPE_NC4C`, `FILETYPE_SRV`, `FILETYPE_EXT` and `FILETYPE_IEG`.

4.1.5. Define the byte order: `streamDefByteorder`

The function `streamDefByteorder` defines the byte order of a binary dataset with the file format type `FILETYPE_SRV`, `FILETYPE_EXT` or `FILETYPE_IEG`.

Usage

```
SUBROUTINE streamDefByteorder(INTEGER streamID, INTEGER byteorder)
```

`streamID` Stream ID, from a previous call to `streamOpenWrite`.

`byteorder` The byte order of a dataset, one of the **CDI** constants `CDI_BIGENDIAN` and `CDI_LITTLEENDIAN`.

4.1.6. Get the byte order: `streamInqByteorder`

The function `streamInqByteorder` returns the byte order of a binary dataset with the file format type `FILETYPE_SRV`, `FILETYPE_EXT` or `FILETYPE_IEG`.

Usage

```
INTEGER FUNCTION streamInqByteorder(INTEGER streamID)
```

`streamID` Stream ID, from a previous call to `streamOpenRead` or `streamOpenWrite`.

Result

`streamInqByteorder` returns the type of the byte order. The valid **CDI** byte order types are `CDI_BIGENDIAN` and `CDI_LITTLEENDIAN`

4.1.7. Define the variable list: `streamDefVlist`

The function `streamDefVlist` defines the variable list of a stream.

Usage

```
SUBROUTINE streamDefVlist(INTEGER streamID, INTEGER vlistID)
```

`streamID`	Stream ID, from a previous call to `streamOpenWrite`.
`vlistID`	Variable list ID, from a previous call to `vlistCreate`.

4.1.8. Get the variable list: `streamInqVlist`

The function `streamInqVlist` returns the variable list of a stream.

Usage

```
INTEGER FUNCTION streamInqVlist(INTEGER streamID)
```

`streamID`	Stream ID, from a previous call to `streamOpenRead` or `streamOpenWrite`.

Result

`streamInqVlist` returns an identifier to the variable list.

4.1.9. Define time step: `streamDefTimestep`

The function `streamDefTimestep` defines the time step of a stream.

Usage

```
INTEGER FUNCTION streamDefTimestep(INTEGER streamID, INTEGER tsID)
```

`streamID`	Stream ID, from a previous call to `streamOpenWrite`.
`tsID`	Timestep identifier.

Result

`streamDefTimestep` returns the number of records of the time step.

4.1.10. Get time step: `streamInqTimestep`

The function `streamInqTimestep` returns the time step of a stream.

Usage

```
INTEGER FUNCTION streamInqTimestep(INTEGER streamID, INTEGER tsID)
```

`streamID`	Stream ID, from a previous call to `streamOpenRead` or `streamOpenWrite`.
`tsID`	Timestep identifier.

Result

`streamInqTimestep` returns the number of records of the time step.

4.1.11. Write a variable: `streamWriteVar`

The function streamWriteVar writes the values of one time step of a variable to an open dataset. The values are converted to the external data type of the variable, if necessary.

Usage

```
SUBROUTINE streamWriteVar(INTEGER streamID, INTEGER varID, REAL*8 data,
                          INTEGER nmiss)
```

`streamID`	Stream ID, from a previous call to `streamOpenWrite`.
`varID`	Variable identifier.
`data`	Pointer to a block of double precision floating point data values to be written.
`nmiss`	Number of missing values.

4.1.12. Write a variable: `streamWriteVarF`

The function streamWriteVarF writes the values of one time step of a variable to an open dataset. The values are converted to the external data type of the variable, if necessary. Only support for netCDF was implemented in this function.

Usage

```
SUBROUTINE streamWriteVarF(INTEGER streamID, INTEGER varID, REAL*4 data,
                           INTEGER nmiss)
```

`streamID`	Stream ID, from a previous call to `streamOpenWrite`.
`varID`	Variable identifier.
`data`	Pointer to a block of single precision floating point data values to be written.
`nmiss`	Number of missing values.

4.1.13. Read a variable: `streamReadVar`

The function streamReadVar reads all the values of one time step of a variable from an open dataset.

Usage

```
SUBROUTINE streamReadVar(INTEGER streamID, INTEGER varID, REAL*8 data,
                         INTEGER nmiss)
```

`streamID`	Stream ID, from a previous call to `streamOpenRead`.
`varID`	Variable identifier.
`data`	Pointer to the location into which the data values are read. The caller must allocate space for the returned values.
`nmiss`	Number of missing values.

4.1.14. Write a horizontal slice of a variable: `streamWriteVarSlice`

The function streamWriteVarSlice writes the values of a horizontal slice of a variable to an open dataset. The values are converted to the external data type of the variable, if necessary.

Usage

```
SUBROUTINE streamWriteVarSlice(INTEGER streamID, INTEGER varID, INTEGER levelID,
                               REAL*8 data, INTEGER nmiss)
```

`streamID`	Stream ID, from a previous call to `streamOpenWrite`.
`varID`	Variable identifier.
`levelID`	Level identifier.
`data`	Pointer to a block of double precision floating point data values to be written.
`nmiss`	Number of missing values.

4.1.15. Write a horizontal slice of a variable: `streamWriteVarSliceF`

The function streamWriteVarSliceF writes the values of a horizontal slice of a variable to an open dataset. The values are converted to the external data type of the variable, if necessary. Only support for netCDF was implemented in this function.

Usage

```
SUBROUTINE streamWriteVarSliceF(INTEGER streamID, INTEGER varID, INTEGER levelID,
                                REAL*4 data, INTEGER nmiss)
```

`streamID`	Stream ID, from a previous call to `streamOpenWrite`.
`varID`	Variable identifier.
`levelID`	Level identifier.
`data`	Pointer to a block of single precision floating point data values to be written.
`nmiss`	Number of missing values.

4.1.16. Read a horizontal slice of a variable: `streamReadVarSlice`

The function streamReadVarSlice reads all the values of a horizontal slice of a variable from an open dataset.

Usage

```
SUBROUTINE streamReadVarSlice(INTEGER streamID, INTEGER varID, INTEGER levelID,
                              REAL*8 data, INTEGER nmiss)
```

`streamID`	Stream ID, from a previous call to `streamOpenRead`.
`varID`	Variable identifier.
`levelID`	Level identifier.
`data`	Pointer to the location into which the data values are read. The caller must allocate space for the returned values.
`nmiss`	Number of missing values.

4.2. Variable list functions

This module contains functions to handle a list of variables. A variable list is a collection of all variables of a dataset.

4.2.1. Create a variable list: vlistCreate

Usage

```
INTEGER FUNCTION vlistCreate()
```

Example

Here is an example using `vlistCreate` to create a variable list and add a variable with `vlistDefVar`.

```
INCLUDE 'cdi.h'
   ...
INTEGER vlistID, varID
   ...
vlistID = vlistCreate()
varID = vlistDefVar(vlistID, gridID, zaxisID, TSTEP_INSTANT)
   ...
streamDefVlist(streamID, vlistID)
   ...
vlistDestroy(vlistID)
   ...
```

4.2.2. Destroy a variable list: vlistDestroy

Usage

```
SUBROUTINE vlistDestroy(INTEGER vlistID)
```

`vlistID` Variable list ID, from a previous call to `vlistCreate`.

4.2.3. Copy a variable list: vlistCopy

The function `vlistCopy` copies all entries from vlistID1 to vlistID2.

Usage

```
SUBROUTINE vlistCopy(INTEGER vlistID2, INTEGER vlistID1)
```

`vlistID2` Target variable list ID.

`vlistID1` Source variable list ID.

4.2.4. Duplicate a variable list: vlistDuplicate

The function `vlistDuplicate` duplicates the variable list from vlistID1.

Usage

```
INTEGER FUNCTION vlistDuplicate(INTEGER vlistID)
```

`vlistID` Variable list ID, from a previous call to `vlistCreate` or `streamInqVlist`.

Result

`vlistDuplicate` returns an identifier to the duplicated variable list.

4.2.5. Concatenate two variable lists: `vlistCat`

Concatenate the variable list vlistID1 at the end of vlistID2.

Usage

```
SUBROUTINE vlistCat(INTEGER vlistID2, INTEGER vlistID1)
```

`vlistID2`	Target variable list ID.
`vlistID1`	Source variable list ID.

4.2.6. Copy some entries of a variable list: `vlistCopyFlag`

The function `vlistCopyFlag` copies all entries with a flag from vlistID1 to vlistID2.

Usage

```
SUBROUTINE vlistCopyFlag(INTEGER vlistID2, INTEGER vlistID1)
```

`vlistID2`	Target variable list ID.
`vlistID1`	Source variable list ID.

4.2.7. Number of variables in a variable list: `vlistNvars`

The function `vlistNvars` returns the number of variables in the variable list vlistID.

Usage

```
INTEGER FUNCTION vlistNvars(INTEGER vlistID)
```

`vlistID`	Variable list ID, from a previous call to `vlistCreate` or `streamInqVlist`.

Result

`vlistNvars` returns the number of variables in a variable list.

4.2.8. Number of grids in a variable list: `vlistNgrids`

The function `vlistNgrids` returns the number of grids in the variable list vlistID.

Usage

```
INTEGER FUNCTION vlistNgrids(INTEGER vlistID)
```

`vlistID`	Variable list ID, from a previous call to `vlistCreate` or `streamInqVlist`.

Result

`vlistNgrids` returns the number of grids in a variable list.

4.2.9. Number of zaxis in a variable list: `vlistNzaxis`

The function `vlistNzaxis` returns the number of zaxis in the variable list vlistID.

Usage

```
INTEGER FUNCTION vlistNzaxis(INTEGER vlistID)
```

`vlistID` Variable list ID, from a previous call to `vlistCreate` or `streamInqVlist`.

Result

`vlistNzaxis` returns the number of zaxis in a variable list.

4.2.10. Define the time axis: `vlistDefTaxis`

The function `vlistDefTaxis` defines the time axis of a variable list.

Usage

```
SUBROUTINE vlistDefTaxis(INTEGER vlistID, INTEGER taxisID)
```

`vlistID` Variable list ID, from a previous call to `vlistCreate`.

`taxisID` Time axis ID, from a previous call to `taxisCreate`.

4.2.11. Get the time axis: `vlistInqTaxis`

The function `vlistInqTaxis` returns the time axis of a variable list.

Usage

```
INTEGER FUNCTION vlistInqTaxis(INTEGER vlistID)
```

`vlistID` Variable list ID, from a previous call to `vlistCreate` or `streamInqVlist`.

Result

`vlistInqTaxis` returns an identifier to the time axis.

4.3. Variable functions

This module contains functions to add new variables to a variable list and to get information about variables from a variable list. To add new variables to a variables list one of the following time types must be specified:

`TSTEP_CONSTANT`	For time constant variables
`TSTEP_INSTANT`	For time varying variables

The default data type is 16 bit for GRIB and 32 bit for all other file format types. To change the data type use one of the following predefined constants:

`DATATYPE_PACK8`	8 packed bit (only for GRIB)
`DATATYPE_PACK16`	16 packed bit (only for GRIB)
`DATATYPE_PACK24`	24 packed bit (only for GRIB)
`DATATYPE_FLT32`	32 bit floating point
`DATATYPE_FLT64`	64 bit floating point
`DATATYPE_INT8`	8 bit integer
`DATATYPE_INT16`	16 bit integer
`DATATYPE_INT32`	32 bit integer

4.3.1. Define a Variable: `vlistDefVar`

The function `vlistDefVar` adds a new variable to vlistID.

Usage

```
INTEGER FUNCTION vlistDefVar(INTEGER vlistID, INTEGER gridID, INTEGER zaxisID,
                             INTEGER tsteptype)
```

`vlistID`	Variable list ID, from a previous call to `vlistCreate`.
`gridID`	Grid ID, from a previous call to `gridCreate`.
`zaxisID`	Z-axis ID, from a previous call to `zaxisCreate`.
`tsteptype`	One of the set of predefined **CDI** timestep types. The valid **CDI** timestep types are `TSTEP_CONSTANT` and `TSTEP_INSTANT`.

Result

`vlistDefVar` returns an identifier to the new variable.

Example

Here is an example using `vlistCreate` to create a variable list and add a variable with `vlistDefVar`.

```
INCLUDE 'cdi.h'
   ...
INTEGER vlistID, varID
   ...
vlistID  = vlistCreate()
varID = vlistDefVar(vlistID, gridID, zaxisID, TIME_INSTANT)
   ...
streamDefVlist(streamID, vlistID)
   ...
```

```
    vlistDestroy(vlistID)
       ...
```

4.3.2. Get the Grid ID of a Variable: `vlistInqVarGrid`

The function `vlistInqVarGrid` returns the grid ID of a variable.

Usage

```
INTEGER FUNCTION vlistInqVarGrid(INTEGER vlistID, INTEGER varID)
```

`vlistID`	Variable list ID, from a previous call to `vlistCreate` or `streamInqVlist`.
`varID`	Variable identifier.

Result

`vlistInqVarGrid` returns the grid ID of the variable.

4.3.3. Get the Zaxis ID of a Variable: `vlistInqVarZaxis`

The function `vlistInqVarZaxis` returns the zaxis ID of a variable.

Usage

```
INTEGER FUNCTION vlistInqVarZaxis(INTEGER vlistID, INTEGER varID)
```

`vlistID`	Variable list ID, from a previous call to `vlistCreate` or `streamInqVlist`.
`varID`	Variable identifier.

Result

`vlistInqVarZaxis` returns the zaxis ID of the variable.

4.3.4. Define the code number of a Variable: `vlistDefVarCode`

The function `vlistDefVarCode` defines the code number of a variable.

Usage

```
SUBROUTINE vlistDefVarCode(INTEGER vlistID, INTEGER varID, INTEGER code)
```

`vlistID`	Variable list ID, from a previous call to `vlistCreate`.
`varID`	Variable identifier.
`code`	Code number.

4.3.5. Get the Code number of a Variable: `vlistInqVarCode`

The function `vlistInqVarCode` returns the code number of a variable.

Usage

```
INTEGER FUNCTION vlistInqVarCode(INTEGER vlistID, INTEGER varID)
```

`vlistID`	Variable list ID, from a previous call to `vlistCreate` or `streamInqVlist`.
`varID`	Variable identifier.

Result

`vlistInqVarCode` returns the code number of the variable.

4.3.6. Define the name of a Variable: `vlistDefVarName`

The function `vlistDefVarName` defines the name of a variable.

Usage

```
SUBROUTINE vlistDefVarName(INTEGER vlistID, INTEGER varID, CHARACTER*(*) name)
```

`vlistID`	Variable list ID, from a previous call to `vlistCreate`.
`varID`	Variable identifier.
`name`	Name of the variable.

4.3.7. Get the name of a Variable: `vlistInqVarName`

The function `vlistInqVarName` returns the name of a variable.

Usage

```
SUBROUTINE vlistInqVarName(INTEGER vlistID, INTEGER varID, CHARACTER*(*) name)
```

`vlistID`	Variable list ID, from a previous call to `vlistCreate` or `streamInqVlist`.
`varID`	Variable identifier.
`name`	Returned variable name. The caller must allocate space for the returned string. The maximum possible length, in characters, of the string is given by the predefined constant `CDI_MAX_NAME`.

Result

`vlistInqVarName` returns the name of the variable to the parameter name if available, otherwise the result is an empty string.

4.3.8. Define the long name of a Variable: `vlistDefVarLongname`

The function `vlistDefVarLongname` defines the long name of a variable.

Usage

```
SUBROUTINE vlistDefVarLongname(INTEGER vlistID, INTEGER varID,
                               CHARACTER*(*) longname)
```

`vlistID`	Variable list ID, from a previous call to `vlistCreate`.
`varID`	Variable identifier.
`longname`	Long name of the variable.

4.3.9. Get the longname of a Variable: `vlistInqVarLongname`

The function `vlistInqVarLongname` returns the longname of a variable if available, otherwise the result is an empty string.

Usage

```
SUBROUTINE vlistInqVarLongname(INTEGER vlistID, INTEGER varID,
                               CHARACTER*(*) longname)
```

`vlistID` Variable list ID, from a previous call to `vlistCreate` or `streamInqVlist`.

`varID` Variable identifier.

`longname` Long name of the variable. The caller must allocate space for the returned string. The maximum possible length, in characters, of the string is given by the predefined constant `CDI_MAX_NAME`.

Result

`vlistInqVaeLongname` returns the longname of the variable to the parameter longname.

4.3.10. Define the standard name of a Variable: `vlistDefVarStdname`

The function `vlistDefVarStdname` defines the standard name of a variable.

Usage

```
SUBROUTINE vlistDefVarStdname(INTEGER vlistID, INTEGER varID,
                              CHARACTER*(*) stdname)
```

`vlistID` Variable list ID, from a previous call to `vlistCreate`.

`varID` Variable identifier.

`stdname` Standard name of the variable.

4.3.11. Get the standard name of a Variable: `vlistInqVarStdname`

The function `vlistInqVarStdname` returns the standard name of a variable if available, otherwise the result is an empty string.

Usage

```
SUBROUTINE vlistInqVarStdname(INTEGER vlistID, INTEGER varID,
                              CHARACTER*(*) stdname)
```

`vlistID` Variable list ID, from a previous call to `vlistCreate` or `streamInqVlist`.

`varID` Variable identifier.

`stdname` Standard name of the variable. The caller must allocate space for the returned string. The maximum possible length, in characters, of the string is given by the predefined constant `CDI_MAX_NAME`.

Result

`vlistInqVarName` returns the standard name of the variable to the parameter stdname.

4.3.12. Define the units of a Variable: `vlistDefVarUnits`

The function `vlistDefVarUnits` defines the units of a variable.

Usage

```
SUBROUTINE vlistDefVarUnits(INTEGER vlistID, INTEGER varID, CHARACTER*(*) units)
```

`vlistID` Variable list ID, from a previous call to `vlistCreate`.

`varID` Variable identifier.

`units` Units of the variable.

4.3.13. Get the units of a Variable: `vlistInqVarUnits`

The function `vlistInqVarUnits` returns the units of a variable if available, otherwise the result is an empty string.

Usage

```
SUBROUTINE vlistInqVarUnits(INTEGER vlistID, INTEGER varID, CHARACTER*(*) units)
```

`vlistID` Variable list ID, from a previous call to `vlistCreate` or `streamInqVlist`.

`varID` Variable identifier.

`units` Units of the variable. The caller must allocate space for the returned string. The maximum possible length, in characters, of the string is given by the predefined constant `CDI_MAX_NAME`.

Result

`vlistInqVarUnits` returns the units of the variable to the parameter units.

4.3.14. Define the data type of a Variable: `vlistDefVarDatatype`

The function `vlistDefVarDatatype` defines the data type of a variable.

Usage

```
SUBROUTINE vlistDefVarDatatype(INTEGER vlistID, INTEGER varID, INTEGER datatype)
```

`vlistID` Variable list ID, from a previous call to `vlistCreate`.

`varID` Variable identifier.

`datatype` The data type identifier. The valid **CDI** data types are `DATATYPE_PACK8`, `DATATYPE_PACK16`, `DATATYPE_PACK24`, `DATATYPE_FLT32`, `DATATYPE_FLT64`, `DATATYPE_INT8`, `DATATYPE_INT16` and `DATATYPE_INT32`.

4.3.15. Get the data type of a Variable: `vlistInqVarDatatype`

The function `vlistInqVarDatatype` returns the data type of a variable.

Usage

```
INTEGER FUNCTION vlistInqVarDatatype(INTEGER vlistID, INTEGER varID)
```

`vlistID` Variable list ID, from a previous call to `vlistCreate` or `streamInqVlist`.

`varID` Variable identifier.

Result

`vlistInqVarDatatype` returns an identifier to the data type of the variable. The valid **CDI** data types are `DATATYPE_PACK8`, `DATATYPE_PACK16`, `DATATYPE_PACK24`, `DATATYPE_FLT32`, `DATATYPE_FLT64`, `DATATYPE_INT8`, `DATATYPE_INT16` and `DATATYPE_INT32`.

4.3.16. Define the missing value of a Variable: `vlistDefVarMissval`

The function `vlistDefVarMissval` defines the missing value of a variable.

Usage

```
SUBROUTINE vlistDefVarMissval(INTEGER vlistID, INTEGER varID, REAL*8 missval)
```

`vlistID` Variable list ID, from a previous call to `vlistCreate`.

`varID` Variable identifier.

`missval` Missing value.

4.3.17. Get the missing value of a Variable: `vlistInqVarMissval`

The function `vlistInqVarMissval` returns the missing value of a variable.

Usage

```
REAL*8 FUNCTION vlistInqVarMissval(INTEGER vlistID, INTEGER varID)
```

`vlistID` Variable list ID, from a previous call to `vlistCreate` or `streamInqVlist`.

`varID` Variable identifier.

Result

`vlistInqVarMissval` returns the missing value of the variable.

4.4. Attributes

Attributes may be associated with each variable to specify non CDI standard properties. CDI standard properties as code, name, units, and missing value are directly associated with each variable by the corresponding CDI function (e.g. vlistDefVarName). An attribute has a variable to which it is assigned, a name, a type, a length, and a sequence of one or more values. The attributes have to be defined after the variable is created and before the variable list is associated with a stream. Attributes are only used for netCDF datasets.
It is also possible to have attributes that are not associated with any variable. These are called global attributes and are identified by using CDI_GLOBAL as a variable pseudo-ID. Global attributes are usually related to the dataset as a whole.
CDI supports integer, floating point and text attributes. The data types are defined by the following predefined constants:

`DATATYPE_INT16`	16-bit integer attribute
`DATATYPE_INT32`	32-bit integer attribute
`DATATYPE_FLT32`	32-bit floating point attribute
`DATATYPE_FLT64`	64-bit floating point attribute
`DATATYPE_TXT`	Text attribute

4.4.1. Get number of variable attributes: `vlistInqNatts`

The function `vlistInqNatts` gets the number of variable attributes assigned to this variable.

Usage

```
INTEGER FUNCTION vlistInqNatts(INTEGER vlistID, INTEGER varID, INTEGER nattsp)
```

`vlistID`	Variable list ID, from a previous call to vlistCreate or streamInqVlist.
`varID`	Variable identifier, or `CDI_GLOBAL` for a global attribute.
`nattsp`	Pointer to location for returned number of variable attributes.

4.4.2. Get information about an attribute: `vlistInqAtt`

The function `vlistInqAtt` gets information about an attribute.

Usage

```
INTEGER FUNCTION vlistInqAtt(INTEGER vlistID, INTEGER varID, INTEGER attnum,
                             CHARACTER*(*) name, INTEGER typep, INTEGER lenp)
```

`vlistID`	Variable list ID, from a previous call to vlistCreate or streamInqVlist.
`varID`	Variable identifier, or `CDI_GLOBAL` for a global attribute.
`attnum`	Attribute number (from 0 to natts-1).
`name`	Pointer to the location for the returned attribute name. The caller must allocate space for the returned string. The maximum possible length, in characters, of the string is given by the predefined constant `CDI_MAX_NAME`.
`typep`	Pointer to location for returned attribute type.
`lenp`	Pointer to location for returned attribute number.

4.4.3. Define an integer attribute: `vlistDefAttInt`

The function `vlistDefAttInt` defines an integer attribute.

Usage

```
INTEGER FUNCTION vlistDefAttInt(INTEGER vlistID, INTEGER varID,
                                CHARACTER*(*) name, INTEGER type, INTEGER len,
                                INTEGER ip)
```

`vlistID`	Variable list ID, from a previous call to `vlistCreate`.
`varID`	Variable identifier, or `CDI_GLOBAL` for a global attribute.
`name`	Attribute name.
`type`	External data type (`DATATYPE_INT16` or `DATATYPE_INT32`).
`len`	Number of values provided for the attribute.
`ip`	Pointer to one or more integer values.

4.4.4. Get the value(s) of an integer attribute: `vlistInqAttInt`

The function `vlistInqAttInt` gets the values(s) of an integer attribute.

Usage

```
INTEGER FUNCTION vlistInqAttInt(INTEGER vlistID, INTEGER varID,
                                CHARACTER*(*) name, INTEGER mlen, INTEGER ip)
```

`vlistID`	Variable list ID, from a previous call to `vlistCreate` or `streamInqVlist`.
`varID`	Variable identifier, or `CDI_GLOBAL` for a global attribute.
`name`	Attribute name.
`mlen`	Number of allocated values provided for the attribute.
`ip`	Pointer location for returned integer attribute value(s).

4.4.5. Define a floating point attribute: `vlistDefAttFlt`

The function `vlistDefAttFlt` defines a floating point attribute.

Usage

```
INTEGER FUNCTION vlistDefAttFlt(INTEGER vlistID, INTEGER varID,
                                CHARACTER*(*) name, INTEGER type, INTEGER len,
                                REAL*8 dp)
```

`vlistID`	Variable list ID, from a previous call to `vlistCreate`.
`varID`	Variable identifier, or `CDI_GLOBAL` for a global attribute.
`name`	Attribute name.
`type`	External data type (`DATATYPE_FLT32` or `DATATYPE_FLT64`).
`len`	Number of values provided for the attribute.
`dp`	Pointer to one or more floating point values.

4.4.6. Get the value(s) of a floating point attribute: `vlistInqAttFlt`

The function `vlistInqAttFlt` gets the values(s) of a floating point attribute.

Usage

```
INTEGER FUNCTION vlistInqAttFlt(INTEGER vlistID, INTEGER varID,
                                CHARACTER*(*) name, INTEGER mlen, REAL*8 dp)
```

`vlistID`	Variable list ID, from a previous call to `vlistCreate` or `streamInqVlist`.
`varID`	Variable identifier, or `CDI_GLOBAL` for a global attribute.
`name`	Attribute name.
`mlen`	Number of allocated values provided for the attribute.
`dp`	Pointer location for returned floating point attribute value(s).

4.4.7. Define a text attribute: `vlistDefAttTxt`

The function `vlistDefAttTxt` defines a text attribute.

Usage

```
INTEGER FUNCTION vlistDefAttTxt(INTEGER vlistID, INTEGER varID,
                                CHARACTER*(*) name, INTEGER len,
                                CHARACTER*(*) tp)
```

`vlistID`	Variable list ID, from a previous call to `vlistCreate`.
`varID`	Variable identifier, or `CDI_GLOBAL` for a global attribute.
`name`	Attribute name.
`len`	Number of values provided for the attribute.
`tp`	Pointer to one or more character values.

4.4.8. Get the value(s) of a text attribute: `vlistInqAttTxt`

The function `vlistInqAttTxt` gets the values(s) of a text attribute.

Usage

```
INTEGER FUNCTION vlistInqAttTxt(INTEGER vlistID, INTEGER varID,
                                CHARACTER*(*) name, INTEGER mlen,
                                CHARACTER*(*) tp)
```

`vlistID`	Variable list ID, from a previous call to `vlistCreate` or `streamInqVlist`.
`varID`	Variable identifier, or `CDI_GLOBAL` for a global attribute.
`name`	Attribute name.
`mlen`	Number of allocated values provided for the attribute.
`tp`	Pointer location for returned text attribute value(s).

4.5. Grid functions

This module contains functions to define a new horizontal Grid and to get information from an existing Grid. A Grid object is necessary to define a variable. The following different Grid types are available:

GRID_GENERIC	Generic user defined grid
GRID_LONLAT	Regular longitude/latitude grid
GRID_GAUSSIAN	Regular Gaussian lon/lat grid
GRID_SPECTRAL	Spherical harmonic coefficients
GRID_GME	Icosahedral-hexagonal GME grid
GRID_CURVILINEAR	Curvilinear grid
GRID_UNSTRUCTURED	Unstructured grid
GRID_LCC	Lambert conformal conic grid

4.5.1. Create a horizontal Grid: `gridCreate`

The function `gridCreate` creates a horizontal Grid.

Usage

```
INTEGER FUNCTION gridCreate(INTEGER gridtype, INTEGER size)
```

`gridtype` The type of the grid, one of the set of predefined **CDI** grid types. The valid **CDI** grid types are `GRID_GENERIC`, `GRID_GAUSSIAN`, `GRID_LONLAT`, `GRID_LCC`, `GRID_SPECTRAL`, `GRID_GME`, `GRID_CURVILINEAR` and `GRID_UNSTRUCTURED` and.

`size` Number of gridpoints.

Result

`gridCreate` returns an identifier to the Grid.

Example

Here is an example using `gridCreate` to create a regular lon/lat Grid:

```
INCLUDE 'cdi.h'
   ...
#define nlon 12
#define nlat  6
   ...
REAL*8 lons(nlon) = (/0, 30, 60, 90, 120, 150, 180, 210, 240, 270, 300, 330/)
REAL*8 lats(nlat) = (/-75, -45, -15, 15, 45, 75/)
INTEGER gridID
   ...
gridID = gridCreate(GRID_LONLAT, nlon*nlat)
CALL gridDefXsize(gridID, nlon)
CALL gridDefYsize(gridID, nlat)
CALL gridDefXvals(gridID, lons)
CALL gridDefYvals(gridID, lats)
   ...
```

4.5.2. Destroy a horizontal Grid: `gridDestroy`

Usage

```
SUBROUTINE gridDestroy(INTEGER gridID)
```

`gridID` Grid ID, from a previous call to `gridCreate`.

4.5.3. Duplicate a horizontal Grid: `gridDuplicate`

The function **`gridDuplicate`** duplicates a horizontal Grid.

Usage

```
INTEGER FUNCTION gridDuplicate(INTEGER gridID)
```

`gridID` Grid ID, from a previous call to `gridCreate` or `vlistInqVarGrid`.

Result

`gridDuplicate` returns an identifier to the duplicated Grid.

4.5.4. Get the type of a Grid: `gridInqType`

The function **`gridInqType`** returns the type of a Grid.

Usage

```
INTEGER FUNCTION gridInqType(INTEGER gridID)
```

`gridID` Grid ID, from a previous call to `gridCreate` or `vlistInqVarGrid`.

Result

`gridInqType` returns the type of the grid, one of the set of predefined **CDI** grid types. The valid **CDI** grid types are `GRID_GENERIC`, `GRID_GAUSSIAN`, `GRID_LONLAT`, `GRID_LCC`, `GRID_SPECTRAL`, `GRID_GME`, `GRID_CURVILINEAR` and `GRID_UNSTRUCTURED`.

4.5.5. Get the size of a Grid: `gridInqSize`

The function **`gridInqSize`** returns the size of a Grid.

Usage

```
INTEGER FUNCTION gridInqSize(INTEGER gridID)
```

`gridID` Grid ID, from a previous call to `gridCreate` or `vlistInqVarGrid`.

Result

`gridInqSize` returns the number of grid points of a Grid.

4.5.6. Define the number of values of a X-axis: `gridDefXsize`

The function **`gridDefXsize`** defines the number of values of a X-axis.

Usage

```
SUBROUTINE gridDefXsize(INTEGER gridID, INTEGER xsize)
```

`gridID` Grid ID, from a previous call to `gridCreate`.
`xsize` Number of values of a X-axis.

4.5.7. Get the number of values of a X-axis: `gridInqXsize`

The function `gridInqXsize` returns the number of values of a X-axis.

Usage

```
INTEGER FUNCTION gridInqXsize(INTEGER gridID)
```

`gridID` Grid ID, from a previous call to `gridCreate` or `vlistInqVarGrid`.

Result

`gridInqXsize` returns the number of values of a X-axis.

4.5.8. Define the number of values of a Y-axis: `gridDefYsize`

The function `gridDefYsize` defines the number of values of a Y-axis.

Usage

```
SUBROUTINE gridDefYsize(INTEGER gridID, INTEGER ysize)
```

`gridID` Grid ID, from a previous call to `gridCreate`.
`ysize` Number of values of a Y-axis.

4.5.9. Get the number of values of a Y-axis: `gridInqYsize`

The function `gridInqYsize` returns the number of values of a Y-axis.

Usage

```
INTEGER FUNCTION gridInqYsize(INTEGER gridID)
```

`gridID` Grid ID, from a previous call to `gridCreate` or `vlistInqVarGrid`.

Result

`gridInqYsize` returns the number of values of a Y-axis.

4.5.10. Define the number of parallels between a pole and the equator: `gridDefNP`

The function `gridDefNP` defines the number of parallels between a pole and the equator of a Gaussian grid.

Usage

```
SUBROUTINE gridDefNP(INTEGER gridID, INTEGER np)
```

`gridID` Grid ID, from a previous call to `gridCreate`.
`np` Number of parallels between a pole and the equator.

4.5.11. Get the number of parallels between a pole and the equator: `gridInqNP`

The function **gridInqNP** returns the number of parallels between a pole and the equator of a Gaussian grid.

Usage

```
INTEGER FUNCTION gridInqNP(INTEGER gridID)
```

`gridID` Grid ID, from a previous call to `gridCreate` or `vlistInqVarGrid`.

Result

`gridInqNP` returns the number of parallels between a pole and the equator.

4.5.12. Define the values of a X-axis: `gridDefXvals`

The function **gridDefXvals** defines all values of the X-axis.

Usage

```
SUBROUTINE gridDefXvals(INTEGER gridID, REAL*8 xvals)
```

`gridID` Grid ID, from a previous call to `gridCreate`.

`xvals` X-values of the grid.

4.5.13. Get all values of a X-axis: `gridInqXvals`

The function **gridInqXvals** returns all values of the X-axis.

Usage

```
INTEGER FUNCTION gridInqXvals(INTEGER gridID, REAL*8 xvals)
```

`gridID` Grid ID, from a previous call to `gridCreate` or `vlistInqVarGrid`.

`xvals` Pointer to the location into which the X-values are read. The caller must allocate space for the returned values.

Result

Upon successful completion `gridInqXvals` returns the number of values and the values are stored in `xvals`. Otherwise, 0 is returned and `xvals` is empty.

4.5.14. Define the values of a Y-axis: `gridDefYvals`

The function **gridDefYvals** defines all values of the Y-axis.

Usage

```
SUBROUTINE gridDefYvals(INTEGER gridID, REAL*8 yvals)
```

`gridID` Grid ID, from a previous call to `gridCreate`.

`yvals` Y-values of the grid.

4.5.15. Get all values of a Y-axis: `gridInqYvals`

The function `gridInqYvals` returns all values of the Y-axis.

Usage

```
INTEGER FUNCTION gridInqYvals(INTEGER gridID, REAL*8 yvals)
```

`gridID` Grid ID, from a previous call to `gridCreate` or `vlistInqVarGrid`.

`yvals` Pointer to the location into which the Y-values are read. The caller must allocate space for the returned values.

Result

Upon successful completion `gridInqYvals` returns the number of values and the values are stored in `yvals`. Otherwise, 0 is returned and `yvals` is empty.

4.5.16. Define the bounds of a X-axis: `gridDefXbounds`

The function `gridDefXbounds` defines all bounds of the X-axis.

Usage

```
SUBROUTINE gridDefXbounds(INTEGER gridID, REAL*8 xbounds)
```

`gridID` Grid ID, from a previous call to `gridCreate`.

`xbounds` X-bounds of the grid.

4.5.17. Get the bounds of a X-axis: `gridInqXbounds`

The function `gridInqXbounds` returns the bounds of the X-axis.

Usage

```
INTEGER FUNCTION gridInqXbounds(INTEGER gridID, REAL*8 xbounds)
```

`gridID` Grid ID, from a previous call to `gridCreate` or `vlistInqVarGrid`.

`xbounds` Pointer to the location into which the X-bounds are read. The caller must allocate space for the returned values.

Result

Upon successful completion `gridInqXbounds` returns the number of bounds and the bounds are stored in `xbounds`. Otherwise, 0 is returned and `xbounds` is empty.

4.5.18. Define the bounds of a Y-axis: `gridDefYbounds`

The function `gridDefYbounds` defines all bounds of the Y-axis.

Usage

```
SUBROUTINE gridDefYbounds(INTEGER gridID, REAL*8 ybounds)
```

`gridID` Grid ID, from a previous call to `gridCreate`.

`ybounds` Y-bounds of the grid.

4.5.19. Get the bounds of a Y-axis: `gridInqYbounds`

The function **`gridInqYbounds`** returns the bounds of the Y-axis.

Usage

```
INTEGER FUNCTION gridInqYbounds(INTEGER gridID, REAL*8 ybounds)
```

`gridID` Grid ID, from a previous call to `gridCreate` or `vlistInqVarGrid`.

`ybounds` Pointer to the location into which the Y-bounds are read. The caller must allocate space for the returned values.

Result

Upon successful completion **`gridInqYbounds`** returns the number of bounds and the bounds are stored in **`ybounds`**. Otherwise, 0 is returned and **`ybounds`** is empty.

4.5.20. Define the name of a X-axis: `gridDefXname`

The function **`gridDefXname`** defines the name of a X-axis.

Usage

```
SUBROUTINE gridDefXname(INTEGER gridID, CHARACTER*(*) name)
```

`gridID` Grid ID, from a previous call to `gridCreate`.

`name` Name of the X-axis.

4.5.21. Get the name of a X-axis: `gridInqXname`

The function **`gridInqXname`** returns the name of a X-axis.

Usage

```
SUBROUTINE gridInqXname(INTEGER gridID, CHARACTER*(*) name)
```

`gridID` Grid ID, from a previous call to `gridCreate` or `vlistInqVarGrid`.

`name` Name of the X-axis. The caller must allocate space for the returned string. The maximum possible length, in characters, of the string is given by the predefined constant `CDI_MAX_NAME`.

Result

`gridInqXname` returns the name of the X-axis to the parameter name.

4.5.22. Define the longname of a X-axis: `gridDefXlongname`

The function **`gridDefXlongname`** defines the longname of a X-axis.

Usage

```
SUBROUTINE gridDefXlongname(INTEGER gridID, CHARACTER*(*) longname)
```

`gridID` Grid ID, from a previous call to `gridCreate`.

`longname` Longname of the X-axis.

4.5.23. Get the longname of a X-axis: `gridInqXlongname`

The function `gridInqXlongname` returns the longname of a X-axis.

Usage

```
SUBROUTINE gridInqXlongname(INTEGER gridID, CHARACTER*(*) longname)
```

`gridID` Grid ID, from a previous call to `gridCreate` or `vlistInqVarGrid`.

`longname` Longname of the X-axis. The caller must allocate space for the returned string. The maximum possible length, in characters, of the string is given by the predefined constant `CDI_MAX_NAME`.

Result

`gridInqXlongname` returns the longname of the X-axis to the parameter longname.

4.5.24. Define the units of a X-axis: `gridDefXunits`

The function `gridDefXunits` defines the units of a X-axis.

Usage

```
SUBROUTINE gridDefXunits(INTEGER gridID, CHARACTER*(*) units)
```

`gridID` Grid ID, from a previous call to `gridCreate`.

`units` Units of the X-axis.

4.5.25. Get the units of a X-axis: `gridInqXunits`

The function `gridInqXunits` returns the units of a X-axis.

Usage

```
SUBROUTINE gridInqXunits(INTEGER gridID, CHARACTER*(*) units)
```

`gridID` Grid ID, from a previous call to `gridCreate` or `vlistInqVarGrid`.

`units` Units of the X-axis. The caller must allocate space for the returned string. The maximum possible length, in characters, of the string is given by the predefined constant `CDI_MAX_NAME`.

Result

`gridInqXunits` returns the units of the X-axis to the parameter units.

4.5.26. Define the name of a Y-axis: `gridDefYname`

The function `gridDefYname` defines the name of a Y-axis.

Usage

```
SUBROUTINE gridDefYname(INTEGER gridID, CHARACTER*(*) name)
```

`gridID` Grid ID, from a previous call to `gridCreate`.

`name` Name of the Y-axis.

4.5.27. Get the name of a Y-axis: `gridInqYname`

The function **`gridInqYname`** returns the name of a Y-axis.

Usage

```
SUBROUTINE gridInqYname(INTEGER gridID, CHARACTER*(*) name)
```

`gridID` Grid ID, from a previous call to `gridCreate` or `vlistInqVarGrid`.

`name` Name of the Y-axis. The caller must allocate space for the returned string. The maximum possible length, in characters, of the string is given by the predefined constant `CDI_MAX_NAME`.

Result

`gridInqYname` returns the name of the Y-axis to the parameter name.

4.5.28. Define the longname of a Y-axis: `gridDefYlongname`

The function **`gridDefYlongname`** defines the longname of a Y-axis.

Usage

```
SUBROUTINE gridDefYlongname(INTEGER gridID, CHARACTER*(*) longname)
```

`gridID` Grid ID, from a previous call to `gridCreate`.

`longname` Longname of the Y-axis.

4.5.29. Get the longname of a Y-axis: `gridInqYlongname`

The function **`gridInqYlongname`** returns the longname of a Y-axis.

Usage

```
SUBROUTINE gridInqXlongname(INTEGER gridID, CHARACTER*(*) longname)
```

`gridID` Grid ID, from a previous call to `gridCreate` or `vlistInqVarGrid`.

`longname` Longname of the Y-axis. The caller must allocate space for the returned string. The maximum possible length, in characters, of the string is given by the predefined constant `CDI_MAX_NAME`.

Result

`gridInqYlongname` returns the longname of the Y-axis to the parameter longname.

4.5.30. Define the units of a Y-axis: `gridDefYunits`

The function **`gridDefYunits`** defines the units of a Y-axis.

Usage

```
SUBROUTINE gridDefYunits(INTEGER gridID, CHARACTER*(*) units)
```

`gridID` Grid ID, from a previous call to `gridCreate`.

`units` Units of the Y-axis.

4.5.31. Get the units of a Y-axis: `gridInqYunits`

The function `gridInqYunits` returns the units of a Y-axis.

Usage

```
SUBROUTINE gridInqYunits(INTEGER gridID, CHARACTER*(*) units)
```

`gridID`	Grid ID, from a previous call to `gridCreate` or `vlistInqVarGrid`.
`units`	Units of the Y-axis. The caller must allocate space for the returned string. The maximum possible length, in characters, of the string is given by the predefined constant `CDI_MAX_NAME`.

Result

`gridInqYunits` returns the units of the Y-axis to the parameter units.

4.5.32. Define the reference number for an unstructured grid: `gridDefNumber`

The function `gridDefNumber` defines the reference number for an unstructured grid.

Usage

```
SUBROUTINE gridDefNumber(INTEGER gridID, INTEGER number)
```

`gridID`	Grid ID, from a previous call to `gridCreate`.
`number`	Reference number for an unstructured grid.

4.5.33. Get the reference number to an unstructured grid: `gridInqNumber`

The function `gridInqNumber` returns the reference number to an unstructured grid.

Usage

```
INTEGER FUNCTION gridInqNumber(INTEGER gridID)
```

`gridID`	Grid ID, from a previous call to `gridCreate` or `vlistInqVarGrid`.

Result

`gridInqNumber` returns the reference number to an unstructured grid.

4.5.34. Define the position of grid in the reference file: `gridDefPosition`

The function `gridDefPosition` defines the position of grid in the reference file.

Usage

```
SUBROUTINE gridDefPosition(INTEGER gridID, INTEGER position)
```

`gridID`	Grid ID, from a previous call to `gridCreate`.
`position`	Position of grid in the reference file.

4.5.35. Get the position of grid in the reference file: `gridInqPosition`

The function `gridInqPosition` returns the position of grid in the reference file.

Usage

```
INTEGER FUNCTION gridInqPosition(INTEGER gridID)
```

`gridID` Grid ID, from a previous call to `gridCreate` or `vlistInqVarGrid`.

Result

`gridInqPosition` returns the position of grid in the reference file.

4.5.36. Define the reference URI for an unstructured grid: `gridDefReference`

The function `gridDefReference` defines the reference URI for an unstructured grid.

Usage

```
SUBROUTINE gridDefReference(INTEGER gridID, CHARACTER*(*) reference)
```

`gridID` Grid ID, from a previous call to `gridCreate`.
`reference` Reference URI for an unstructured grid.

4.5.37. Get the reference URI to an unstructured grid: `gridInqReference`

The function `gridInqReference` returns the reference URI to an unstructured grid.

Usage

```
char *gridInqReference(INTEGER gridID, CHARACTER*(*) reference)
```

`gridID` Grid ID, from a previous call to `gridCreate` or `vlistInqVarGrid`.

Result

`gridInqReference` returns the reference URI to an unstructured grid.

4.5.38. Define the UUID for an unstructured grid: `gridDefUUID`

The function `gridDefUUID` defines the UUID for an unstructured grid.

Usage

```
SUBROUTINE gridDefUUID(INTEGER gridID, CHARACTER*(*) uuid)
```

`gridID` Grid ID, from a previous call to `gridCreate`.
`uuid` UUID for an unstructured grid.

4.5.39. Get the UUID to an unstructured grid: `gridInqUUID`

The function `gridInqUUID` returns the UUID to an unstructured grid.

Usage

```
SUBROUTINE gridInqUUID(INTEGER gridID, CHARACTER*(*) uuid)
```

`gridID` Grid ID, from a previous call to `gridCreate` or `vlistInqVarGrid`.

Result

`gridInqUUID` returns the UUID to an unstructured grid to the parameter uuid.

4.6. Z-axis functions

This section contains functions to define a new vertical Z-axis and to get information from an existing Z-axis. A Z-axis object is necessary to define a variable. The following different Z-axis types are available:

`ZAXIS_GENERIC`	Generic user defined level
`ZAXIS_SURFACE`	Surface level
`ZAXIS_MEANSEA`	Mean sea level
`ZAXIS_TOA`	Norminal top of atmosphere
`ZAXIS_ATMOSPHERE`	Entire atmosphere
`ZAXIS_SEA_BOTTOM`	Sea bottom
`ZAXIS_ISENTROPIC`	Isentropic (theta) level
`ZAXIS_HYBRID`	Hybrid level
`ZAXIS_SIGMA`	Sigma level
`ZAXIS_PRESSURE`	Isobaric pressure level in Pascal
`ZAXIS_HEIGHT`	Height above ground in meters
`ZAXIS_ALTITUDE`	Altitude above mean sea level in meters
`ZAXIS_CLOUD_BASE`	Cloud base level
`ZAXIS_CLOUD_TOP`	Level of cloud tops
`ZAXIS_ISOTHERM_ZERO`	Level of 0° C isotherm
`ZAXIS_SNOW`	Snow level
`ZAXIS_LAKE_BOTTOM`	Lake or River Bottom
`ZAXIS_SEDIMENT_BOTTOM`	Bottom Of Sediment Layer
`ZAXIS_SEDIMENT_BOTTOM_TA`	Bottom Of Thermally Active Sediment Layer
`ZAXIS_SEDIMENT_BOTTOM_TW`	Bottom Of Sediment Layer Penetrated By Thermal Wave
`ZAXIS_ZAXIS_MIX_LAYER`	Mixing Layer
`ZAXIS_DEPTH_BELOW_SEA`	Depth below sea level in meters
`ZAXIS_DEPTH_BELOW_LAND`	Depth below land surface in centimeters

4.6.1. Create a vertical Z-axis: `zaxisCreate`

The function `zaxisCreate` creates a vertical Z-axis.

Usage

```
INTEGER FUNCTION zaxisCreate(INTEGER zaxistype, INTEGER size)
```

`zaxistype` The type of the Z-axis, one of the set of predefined **CDI** Z-axis types. The valid **CDI** Z-axis types are `ZAXIS_GENERIC`, `ZAXIS_SURFACE`, `ZAXIS_HYBRID`, `ZAXIS_SIGMA`, `ZAXIS_PRESSURE`, `ZAXIS_HEIGHT`, `ZAXIS_ISENTROPIC`, `ZAXIS_ALTITUDE`, `ZAXIS_MEANSEA`, `ZAXIS_TOA`, `ZAXIS_SEA_BOTTOM`, `ZAXIS_ATMOSPHERE`, `ZAXIS_CLOUD_BASE`, `ZAXIS_CLOUD_TOP`, `ZAXIS_ISOTHERM_ZERO`, `ZAXIS_SNOW`, `ZAXIS_LAKE_BOTTOM`, `ZAXIS_SEDIMENT_BOTTOM`, `ZAXIS_SEDIMENT_BOTTOM_TA`, `ZAXIS_SEDIMENT_BOTTOM_TW`, `ZAXIS_MIX_LAYER`, `ZAXIS_DEPTH_BELOW_SEA` and `ZAXIS_DEPTH_BELOW_LAND`.

`size` Number of levels.

Result

`zaxisCreate` returns an identifier to the Z-axis.

Example

Here is an example using `zaxisCreate` to create a pressure level Z-axis:

```
INCLUDE 'cdi.h'
   ...
#define  nlev     5
   ...
REAL*8 levs(nlev) = (/101300, 92500, 85000, 50000, 20000/)
INTEGER zaxisID
   ...
zaxisID = zaxisCreate(ZAXIS_PRESSURE, nlev)
CALL zaxisDefLevels(zaxisID, levs)
   ...
```

4.6.2. Destroy a vertical Z-axis: `zaxisDestroy`

Usage

```
SUBROUTINE zaxisDestroy(INTEGER zaxisID)
```

`zaxisID` Z-axis ID, from a previous call to `zaxisCreate`.

4.6.3. Get the type of a Z-axis: `zaxisInqType`

The function `zaxisInqType` returns the type of a Z-axis.

Usage

```
INTEGER FUNCTION zaxisInqType(INTEGER zaxisID)
```

`zaxisID` Z-axis ID, from a previous call to `zaxisCreate` or `vlistInqVarZaxis`.

Result

`zaxisInqType` returns the type of the Z-axis, one of the set of predefined **CDI** Z-axis types. The valid **CDI** Z-axis types are `ZAXIS_GENERIC`, `ZAXIS_SURFACE`, `ZAXIS_HYBRID`, `ZAXIS_SIGMA`, `ZAXIS_PRESSURE`, `ZAXIS_HEIGHT`, `ZAXIS_ISENTROPIC`, `ZAXIS_ALTITUDE`, `ZAXIS_MEANSEA`, `ZAXIS_TOA`, `ZAXIS_SEA_BOTTOM`, `ZAXIS_ATMOSPHERE`, `ZAXIS_CLOUD_BASE`, `ZAXIS_CLOUD_TOP`, `ZAXIS_ISOTHERM_ZERO`, `ZAXIS_SNOW`, `ZAXIS_LAKE_BOTTOM`, `ZAXIS_SEDIMENT_BOTTOM`, `ZAXIS_SEDIMENT_BOTTOM_TA`, `ZAXIS_SEDIMENT_BOTTO` `ZAXIS_MIX_LAYER`, `ZAXIS_DEPTH_BELOW_SEA` and `ZAXIS_DEPTH_BELOW_LAND`.

4.6.4. Get the size of a Z-axis: `zaxisInqSize`

The function `zaxisInqSize` returns the size of a Z-axis.

Usage

```
INTEGER FUNCTION zaxisInqSize(INTEGER zaxisID)
```

`zaxisID` Z-axis ID, from a previous call to `zaxisCreate` or `vlistInqVarZaxis`.

Result

`zaxisInqSize` returns the number of levels of a Z-axis.

4.6.5. Define the levels of a Z-axis: `zaxisDefLevels`

The function `zaxisDefLevels` defines the levels of a Z-axis.

Usage

```
SUBROUTINE zaxisDefLevels(INTEGER zaxisID, REAL*8 levels)
```

`zaxisID` Z-axis ID, from a previous call to `zaxisCreate`.

`levels` All levels of the Z-axis.

4.6.6. Get all levels of a Z-axis: `zaxisInqLevels`

The function `zaxisInqLevels` returns all levels of a Z-axis.

Usage

```
SUBROUTINE zaxisInqLevels(INTEGER zaxisID, REAL*8 levels)
```

`zaxisID` Z-axis ID, from a previous call to `zaxisCreate` or `vlistInqVarZaxis`.

`levels` Pointer to the location into which the levels are read. The caller must allocate space for the returned values.

Result

`zaxisInqLevels` saves all levels to the parameter `levels`.

4.6.7. Get one level of a Z-axis: `zaxisInqLevel`

The function `zaxisInqLevel` returns one level of a Z-axis.

Usage

```
REAL*8 FUNCTION zaxisInqLevel(INTEGER zaxisID, INTEGER levelID)
```

`zaxisID` Z-axis ID, from a previous call to `zaxisCreate` or `vlistInqVarZaxis`.

`levelID` Level index (range: 0 to nlevel-1).

Result

`zaxisInqLevel` returns the level of a Z-axis.

4.6.8. Define the name of a Z-axis: `zaxisDefName`

The function `zaxisDefName` defines the name of a Z-axis.

Usage

```
SUBROUTINE zaxisDefName(INTEGER zaxisID, CHARACTER*(*) name)
```

`zaxisID` Z-axis ID, from a previous call to `zaxisCreate`.

`name` Name of the Z-axis.

4.6.9. Get the name of a Z-axis: `zaxisInqName`

The function **`zaxisInqName`** returns the name of a Z-axis.

Usage

```
SUBROUTINE zaxisInqName(INTEGER zaxisID, CHARACTER*(*) name)
```

`zaxisID`	Z-axis ID, from a previous call to `zaxisCreate` or `vlistInqVarZaxis`.
`name`	Name of the Z-axis. The caller must allocate space for the returned string. The maximum possible length, in characters, of the string is given by the predefined constant `CDI_MAX_NAME`.

Result

`zaxisInqName` returns the name of the Z-axis to the parameter name.

4.6.10. Define the longname of a Z-axis: `zaxisDefLongname`

The function **`zaxisDefLongname`** defines the longname of a Z-axis.

Usage

```
SUBROUTINE zaxisDefLongname(INTEGER zaxisID, CHARACTER*(*) longname)
```

`zaxisID`	Z-axis ID, from a previous call to `zaxisCreate`.
`longname`	Longname of the Z-axis.

4.6.11. Get the longname of a Z-axis: `zaxisInqLongname`

The function **`zaxisInqLongname`** returns the longname of a Z-axis.

Usage

```
SUBROUTINE zaxisInqLongname(INTEGER zaxisID, CHARACTER*(*) longname)
```

`zaxisID`	Z-axis ID, from a previous call to `zaxisCreate` or `vlistInqVarZaxis`.
`longname`	Longname of the Z-axis. The caller must allocate space for the returned string. The maximum possible length, in characters, of the string is given by the predefined constant `CDI_MAX_NAME`.

Result

`zaxisInqLongname` returns the longname of the Z-axis to the parameter longname.

4.6.12. Define the units of a Z-axis: `zaxisDefUnits`

The function **`zaxisDefUnits`** defines the units of a Z-axis.

Usage

```
SUBROUTINE zaxisDefUnits(INTEGER zaxisID, CHARACTER*(*) units)
```

`zaxisID`	Z-axis ID, from a previous call to `zaxisCreate`.
`units`	Units of the Z-axis.

4.6.13. Get the units of a Z-axis: `zaxisInqUnits`

The function `zaxisInqUnits` returns the units of a Z-axis.

Usage

```
SUBROUTINE zaxisInqUnits(INTEGER zaxisID, CHARACTER*(*) units)
```

`zaxisID` Z-axis ID, from a previous call to `zaxisCreate` or `vlistInqVarZaxis`.

`units` Units of the Z-axis. The caller must allocate space for the returned string. The maximum possible length, in characters, of the string is given by the predefined constant `CDI_MAX_NAME`.

Result

`zaxisInqUnits` returns the units of the Z-axis to the parameter units.

4.7. T-axis functions

This section contains functions to define a new Time axis and to get information from an existing T-axis. A T-axis object is necessary to define the time axis of a dataset and must be assiged to a variable list using `vlistDefTaxis`. The following different Time axis types are available:

`TAXIS_ABSOLUTE`	Absolute time axis
`TAXIS_RELATIVE`	Relative time axis

An absolute time axis has the current time to each time step. It can be used without knowledge of the calendar.
A relative time is the time relative to a fixed reference time. The current time results from the reference time and the elapsed interval. The result depends on the used calendar. CDI supports the following calendar types:

`CALENDAR_STANDARD`	Mixed Gregorian/Julian calendar.
`CALENDAR_PROLEPTIC`	Proleptic Gregorian calendar. This is the default.
`CALENDAR_360DAYS`	All years are 360 days divided into 30 day months.
`CALENDAR_365DAYS`	Gregorian calendar without leap years, i.e., all years are 365 days long.
`CALENDAR_366DAYS`	Gregorian calendar with every year being a leap year, i.e., all years are 366 days long.

4.7.1. Create a Time axis: `taxisCreate`

The function `taxisCreate` creates a Time axis.

Usage

```
INTEGER FUNCTION taxisCreate(INTEGER taxistype)
```

`taxistype` The type of the Time axis, one of the set of predefined **CDI** time axis types. The valid **CDI** time axis types are `TAXIS_ABSOLUTE` and `TAXIS_RELATIVE`.

Result

`taxisCreate` returns an identifier to the Time axis.

Example

Here is an example using `taxisCreate` to create a relative T-axis with a standard calendar.

```
INCLUDE 'cdi.h'
   ...
INTEGER taxisID
   ...
taxisID = taxisCreate(TAXIS_RELATIVE)
taxisDefCalendar(taxisID, CALENDAR_STANDARD)
taxisDefRdate(taxisID, 19850101)
taxisDefRtime(taxisID, 120000)
   ...
```

4.7.2. Destroy a Time axis: `taxisDestroy`

Usage

```
SUBROUTINE taxisDestroy(INTEGER taxisID)
```

`taxisID` Time axis ID, from a previous call to `taxisCreate`

4.7.3. Define the reference date: `taxisDefRdate`

The function `taxisDefRdate` defines the reference date of a Time axis.

Usage

```
SUBROUTINE taxisDefRdate(INTEGER taxisID, INTEGER rdate)
```

`taxisID` Time axis ID, from a previous call to `taxisCreate`
`rdate` Reference date (YYYYMMDD)

4.7.4. Get the reference date: `taxisInqRdate`

The function `taxisInqRdate` returns the reference date of a Time axis.

Usage

```
INTEGER FUNCTION taxisInqRdate(INTEGER taxisID)
```

`taxisID` Time axis ID, from a previous call to `taxisCreate` or `vlistInqTaxis`

Result

`taxisInqRdate` returns the reference date.

4.7.5. Define the reference time: `taxisDefRtime`

The function `taxisDefRtime` defines the reference time of a Time axis.

Usage

```
SUBROUTINE taxisDefRtime(INTEGER taxisID, INTEGER rtime)
```

`taxisID` Time axis ID, from a previous call to `taxisCreate`
`rtime` Reference time (hhmmss)

4.7.6. Get the reference time: `taxisInqRtime`

The function `taxisInqRtime` returns the reference time of a Time axis.

Usage

```
INTEGER FUNCTION taxisInqRtime(INTEGER taxisID)
```

`taxisID` Time axis ID, from a previous call to `taxisCreate` or `vlistInqTaxis`

Result

`taxisInqRtime` returns the reference time.

4.7.7. Define the verification date: `taxisDefVdate`

The function `taxisDefVdate` defines the verification date of a Time axis.

Usage

```
SUBROUTINE taxisDefVdate(INTEGER taxisID, INTEGER vdate)
```

`taxisID`	Time axis ID, from a previous call to `taxisCreate`
`vdate`	Verification date (YYYYMMDD)

4.7.8. Get the verification date: `taxisInqVdate`

The function `taxisInqVdate` returns the verification date of a Time axis.

Usage

```
INTEGER FUNCTION taxisInqVdate(INTEGER taxisID)
```

`taxisID`	Time axis ID, from a previous call to `taxisCreate` or `vlistInqTaxis`

Result

`taxisInqVdate` returns the verification date.

4.7.9. Define the verification time: `taxisDefVtime`

The function `taxisDefVtime` defines the verification time of a Time axis.

Usage

```
SUBROUTINE taxisDefVtime(INTEGER taxisID, INTEGER vtime)
```

`taxisID`	Time axis ID, from a previous call to `taxisCreate`
`vtime`	Verification time (hhmmss)

4.7.10. Get the verification time: `taxisInqVtime`

The function `taxisInqVtime` returns the verification time of a Time axis.

Usage

```
INTEGER FUNCTION taxisInqVtime(INTEGER taxisID)
```

`taxisID`	Time axis ID, from a previous call to `taxisCreate` or `vlistInqTaxis`

Result

`taxisInqVtime` returns the verification time.

4.7.11. Define the calendar: `taxisDefCalendar`

The function `taxisDefCalendar` defines the calendar of a Time axis.

Usage

```
SUBROUTINE taxisDefCalendar(INTEGER taxisID, INTEGER calendar)
```

`taxisID` Time axis ID, from a previous call to `taxisCreate`

`calendar` The type of the calendar, one of the set of predefined **CDI** calendar types. The valid **CDI** calendar types are `CALENDAR_STANDARD`, `CALENDAR_PROLEPTIC`, `CALENDAR_360DAYS`, `CALENDAR_365DAYS` and `CALENDAR_366DAYS`.

4.7.12. Get the calendar: `taxisInqCalendar`

The function `taxisInqCalendar` returns the calendar of a Time axis.

Usage

```
INTEGER FUNCTION taxisInqCalendar(INTEGER taxisID)
```

`taxisID` Time axis ID, from a previous call to `taxisCreate` or `vlistInqTaxis`

Result

`taxisInqCalendar` returns the type of the calendar, one of the set of predefined **CDI** calendar types. The valid **CDI** calendar types are `CALENDAR_STANDARD`, `CALENDAR_PROLEPTIC`, `CALENDAR_360DAYS`, `CALENDAR_365DAYS` and `CALENDAR_366DAYS`.

Bibliography

[ECHAM]
The atmospheric general circulation model ECHAM5, from the Max Planck Institute for Meteorologie

[GRIB]
GRIB version 1, from the World Meteorological Organisation (WMO)

[GRIBAPI]
GRIB API decoding/encoding, from the European Centre for Medium-Range Weather Forecasts (ECMWF)

[HDF5]
HDF version 5, from the HDF Group

[NetCDF]
NetCDF Software Package, from the UNIDATA Program Center of the University Corporation for Atmospheric Research

[MPIOM]
The ocean model MPIOM, from the Max Planck Institute for Meteorologie

[REMO]
The regional climate model REMO, from the Max Planck Institute for Meteorologie

A. Quick Reference

This appendix provide a brief listing of the Fortran language bindings of the **CDI** library routines:

gridCreate

```
INTEGER FUNCTION gridCreate(INTEGER gridtype, INTEGER size)
```

Create a horizontal Grid

gridDefNP

```
SUBROUTINE gridDefNP(INTEGER gridID, INTEGER np)
```

Define the number of parallels between a pole and the equator

gridDefNumber

```
SUBROUTINE gridDefNumber(INTEGER gridID, INTEGER number)
```

Define the reference number for an unstructured grid

gridDefPosition

```
SUBROUTINE gridDefPosition(INTEGER gridID, INTEGER position)
```

Define the position of grid in the reference file

gridDefReference

```
SUBROUTINE gridDefReference(INTEGER gridID, CHARACTER*(*) reference)
```

Define the reference URI for an unstructured grid

gridDefUUID

```
SUBROUTINE gridDefUUID(INTEGER gridID, CHARACTER*(*) uuid)
```

Define the UUID for an unstructured grid

gridDefXbounds

```
SUBROUTINE gridDefXbounds(INTEGER gridID, REAL*8 xbounds)
```

Define the bounds of a X-axis

gridDefXlongname

```
SUBROUTINE gridDefXlongname(INTEGER gridID, CHARACTER*(*) longname)
```

Define the longname of a X-axis

gridDefXname

```
SUBROUTINE gridDefXname(INTEGER gridID, CHARACTER*(*) name)
```

Define the name of a X-axis

gridDefXsize

```
SUBROUTINE gridDefXsize(INTEGER gridID, INTEGER xsize)
```

Define the number of values of a X-axis

gridDefXunits

```
SUBROUTINE gridDefXunits(INTEGER gridID, CHARACTER*(*) units)
```

Define the units of a X-axis

gridDefXvals

```
SUBROUTINE gridDefXvals(INTEGER gridID, REAL*8 xvals)
```

Define the values of a X-axis

gridDefYbounds

```
SUBROUTINE gridDefYbounds(INTEGER gridID, REAL*8 ybounds)
```

Define the bounds of a Y-axis

gridDefYlongname

```
SUBROUTINE gridDefYlongname(INTEGER gridID, CHARACTER*(*) longname)
```

Define the longname of a Y-axis

gridDefYname

```
SUBROUTINE gridDefYname(INTEGER gridID, CHARACTER*(*) name)
```

Define the name of a Y-axis

gridDefYsize

```
SUBROUTINE gridDefYsize(INTEGER gridID, INTEGER ysize)
```

Define the number of values of a Y-axis

gridDefYunits

```
SUBROUTINE gridDefYunits(INTEGER gridID, CHARACTER*(*) units)
```

Define the units of a Y-axis

gridDefYvals

```
SUBROUTINE gridDefYvals(INTEGER gridID, REAL*8 yvals)
```

Define the values of a Y-axis

gridDestroy

```
SUBROUTINE gridDestroy(INTEGER gridID)
```

Destroy a horizontal Grid

gridDuplicate

```
INTEGER FUNCTION gridDuplicate(INTEGER gridID)
```

Duplicate a horizontal Grid

gridInqNP

```
INTEGER FUNCTION gridInqNP(INTEGER gridID)
```

Get the number of parallels between a pole and the equator

gridInqNumber

```
INTEGER FUNCTION gridInqNumber(INTEGER gridID)
```

Get the reference number to an unstructured grid

gridInqPosition

```
INTEGER FUNCTION gridInqPosition(INTEGER gridID)
```

Get the position of grid in the reference file

gridInqReference

```
char *gridInqReference(INTEGER gridID, CHARACTER*(*) reference)
```

Get the reference URI to an unstructured grid

gridInqSize

```
INTEGER FUNCTION gridInqSize(INTEGER gridID)
```

Get the size of a Grid

gridInqType

```
INTEGER FUNCTION gridInqType(INTEGER gridID)
```

Get the type of a Grid

gridInqUUID

```
SUBROUTINE gridInqUUID(INTEGER gridID, CHARACTER*(*) uuid)
```

Get the UUID to an unstructured grid

gridInqXbounds

```
INTEGER FUNCTION gridInqXbounds(INTEGER gridID, REAL*8 xbounds)
```

Get the bounds of a X-axis

gridInqXlongname

```
SUBROUTINE gridInqXlongname(INTEGER gridID, CHARACTER*(*) longname)
```

Get the longname of a X-axis

gridInqXname

```
SUBROUTINE gridInqXname(INTEGER gridID, CHARACTER*(*) name)
```

Get the name of a X-axis

gridInqXsize

```
INTEGER FUNCTION gridInqXsize(INTEGER gridID)
```

Get the number of values of a X-axis

gridInqXunits

```
SUBROUTINE gridInqXunits(INTEGER gridID, CHARACTER*(*) units)
```

Get the units of a X-axis

gridInqXvals

```
INTEGER FUNCTION gridInqXvals(INTEGER gridID, REAL*8 xvals)
```

Get all values of a X-axis

gridInqYbounds

```
INTEGER FUNCTION gridInqYbounds(INTEGER gridID, REAL*8 ybounds)
```

Get the bounds of a Y-axis

gridInqYlongname

```
SUBROUTINE gridInqXlongname(INTEGER gridID, CHARACTER*(*) longname)
```

Get the longname of a Y-axis

gridInqYname

```
SUBROUTINE gridInqYname(INTEGER gridID, CHARACTER*(*) name)
```

Get the name of a Y-axis

gridInqYsize

```
INTEGER FUNCTION gridInqYsize(INTEGER gridID)
```

Get the number of values of a Y-axis

gridInqYunits

```
SUBROUTINE gridInqYunits(INTEGER gridID, CHARACTER*(*) units)
```

Get the units of a Y-axis

gridInqYvals

```
INTEGER FUNCTION gridInqYvals(INTEGER gridID, REAL*8 yvals)
```

Get all values of a Y-axis

streamClose

```
SUBROUTINE streamClose(INTEGER streamID)
```

Close an open dataset

streamDefByteorder

```
SUBROUTINE streamDefByteorder(INTEGER streamID, INTEGER byteorder)
```

Define the byte order

streamDefRecord

```
SUBROUTINE streamDefRecord(INTEGER streamID, INTEGER varID, INTEGER levelID)
```

Define the next record

streamDefTimestep

```
INTEGER FUNCTION streamDefTimestep(INTEGER streamID, INTEGER tsID)
```

Define time step

streamDefVlist

```
SUBROUTINE streamDefVlist(INTEGER streamID, INTEGER vlistID)
```

Define the variable list

streamInqByteorder

```
INTEGER FUNCTION streamInqByteorder(INTEGER streamID)
```

Get the byte order

streamInqFiletype

```
INTEGER FUNCTION streamInqFiletype(INTEGER streamID)
```

Get the filetype

streamInqTimestep

```
INTEGER FUNCTION streamInqTimestep(INTEGER streamID, INTEGER tsID)
```

Get time step

streamInqVlist

```
INTEGER FUNCTION streamInqVlist(INTEGER streamID)
```

Get the variable list

streamOpenRead

```
INTEGER FUNCTION streamOpenRead(CHARACTER*(*) path)
```

Open a dataset for reading

streamOpenWrite

```
INTEGER FUNCTION streamOpenWrite(CHARACTER*(*) path, INTEGER filetype)
```

Create a new dataset

streamReadVar

```
SUBROUTINE streamReadVar(INTEGER streamID, INTEGER varID, REAL*8 data,
                         INTEGER nmiss)
```

Read a variable

streamReadVarSlice

```
SUBROUTINE streamReadVarSlice(INTEGER streamID, INTEGER varID, INTEGER levelID,
                              REAL*8 data, INTEGER nmiss)
```

Read a horizontal slice of a variable

streamWriteRecord

```
SUBROUTINE streamWriteRecord(INTEGER streamID, REAL*8 data, INTEGER nmiss)
```

Write a horizontal slice of a variable

streamWriteVar

```
SUBROUTINE streamWriteVar(INTEGER streamID, INTEGER varID, REAL*8 data,
                          INTEGER nmiss)
```

Write a variable

streamWriteVarF

```
SUBROUTINE streamWriteVarF(INTEGER streamID, INTEGER varID, REAL*4 data,
                           INTEGER nmiss)
```

Write a variable

streamWriteVarSlice

```
SUBROUTINE streamWriteVarSlice(INTEGER streamID, INTEGER varID, INTEGER levelID,
                               REAL*8 data, INTEGER nmiss)
```

Write a horizontal slice of a variable

streamWriteVarSliceF

```
SUBROUTINE streamWriteVarSliceF(INTEGER streamID, INTEGER varID, INTEGER levelID,
                                REAL*4 data, INTEGER nmiss)
```

Write a horizontal slice of a variable

taxisCreate

```
INTEGER FUNCTION taxisCreate(INTEGER taxistype)
```

Create a Time axis

taxisDefCalendar

```
SUBROUTINE taxisDefCalendar(INTEGER taxisID, INTEGER calendar)
```

Define the calendar

taxisDefRdate

```
SUBROUTINE taxisDefRdate(INTEGER taxisID, INTEGER rdate)
```

Define the reference date

taxisDefRtime

```
SUBROUTINE taxisDefRtime(INTEGER taxisID, INTEGER rtime)
```

Define the reference time

taxisDefVdate

```
SUBROUTINE taxisDefVdate(INTEGER taxisID, INTEGER vdate)
```

Define the verification date

taxisDefVtime

```
SUBROUTINE taxisDefVtime(INTEGER taxisID, INTEGER vtime)
```

Define the verification time

taxisDestroy

```
SUBROUTINE taxisDestroy(INTEGER taxisID)
```

Destroy a Time axis

taxisInqCalendar

```
INTEGER FUNCTION taxisInqCalendar(INTEGER taxisID)
```

Get the calendar

taxisInqRdate

```
INTEGER FUNCTION taxisInqRdate(INTEGER taxisID)
```

Get the reference date

taxisInqRtime

```
INTEGER FUNCTION taxisInqRtime(INTEGER taxisID)
```

Get the reference time

taxisInqVdate

```
INTEGER FUNCTION taxisInqVdate(INTEGER taxisID)
```

Get the verification date

taxisInqVtime

```
INTEGER FUNCTION taxisInqVtime(INTEGER taxisID)
```

Get the verification time

vlistCat

```
SUBROUTINE vlistCat(INTEGER vlistID2, INTEGER vlistID1)
```

Concatenate two variable lists

vlistCopy

```
SUBROUTINE vlistCopy(INTEGER vlistID2, INTEGER vlistID1)
```

Copy a variable list

vlistCopyFlag

```
SUBROUTINE vlistCopyFlag(INTEGER vlistID2, INTEGER vlistID1)
```

Copy some entries of a variable list

vlistCreate

```
INTEGER FUNCTION vlistCreate()
```

Create a variable list

vlistDefAttFlt

```
INTEGER FUNCTION vlistDefAttFlt(INTEGER vlistID, INTEGER varID,
                                CHARACTER*(*) name, INTEGER type, INTEGER len,
                                REAL*8 dp)
```

Define a floating point attribute

vlistDefAttInt

```
INTEGER FUNCTION vlistDefAttInt(INTEGER vlistID, INTEGER varID,
                                CHARACTER*(*) name, INTEGER type, INTEGER len,
                                INTEGER ip)
```

Define an integer attribute

vlistDefAttTxt

```
INTEGER FUNCTION vlistDefAttTxt(INTEGER vlistID, INTEGER varID,
                                CHARACTER*(*) name, INTEGER len,
                                CHARACTER*(*) tp)
```

Define a text attribute

vlistDefTaxis

```
SUBROUTINE vlistDefTaxis(INTEGER vlistID, INTEGER taxisID)
```

Define the time axis

vlistDefVar

```
INTEGER FUNCTION vlistDefVar(INTEGER vlistID, INTEGER gridID, INTEGER zaxisID,
                             INTEGER tsteptype)
```

Define a Variable

vlistDefVarCode

```
SUBROUTINE vlistDefVarCode(INTEGER vlistID, INTEGER varID, INTEGER code)
```

Define the code number of a Variable

vlistDefVarDatatype

```
SUBROUTINE vlistDefVarDatatype(INTEGER vlistID, INTEGER varID, INTEGER datatype)
```

Define the data type of a Variable

vlistDefVarLongname

```
SUBROUTINE vlistDefVarLongname(INTEGER vlistID, INTEGER varID,
                               CHARACTER*(*) longname)
```

Define the long name of a Variable

vlistDefVarMissval

```
SUBROUTINE vlistDefVarMissval(INTEGER vlistID, INTEGER varID, REAL*8 missval)
```

Define the missing value of a Variable

vlistDefVarName

```
SUBROUTINE vlistDefVarName(INTEGER vlistID, INTEGER varID, CHARACTER*(*) name)
```

Define the name of a Variable

vlistDefVarStdname

```
SUBROUTINE vlistDefVarStdname(INTEGER vlistID, INTEGER varID,
                              CHARACTER*(*) stdname)
```

Define the standard name of a Variable

vlistDefVarUnits

```
SUBROUTINE vlistDefVarUnits(INTEGER vlistID, INTEGER varID, CHARACTER*(*) units)
```

Define the units of a Variable

vlistDestroy

```
SUBROUTINE vlistDestroy(INTEGER vlistID)
```

Destroy a variable list

vlistDuplicate

```
INTEGER FUNCTION vlistDuplicate(INTEGER vlistID)
```

Duplicate a variable list

vlistInqAtt

```
INTEGER FUNCTION vlistInqAtt(INTEGER vlistID, INTEGER varID, INTEGER attnum,
                             CHARACTER*(*) name, INTEGER typep, INTEGER lenp)
```

Get information about an attribute

vlistInqAttFlt

```
INTEGER FUNCTION vlistInqAttFlt(INTEGER vlistID, INTEGER varID,
                                CHARACTER*(*) name, INTEGER mlen, REAL*8 dp)
```

Get the value(s) of a floating point attribute

vlistInqAttInt

```
INTEGER FUNCTION vlistInqAttInt(INTEGER vlistID, INTEGER varID,
                                CHARACTER*(*) name, INTEGER mlen, INTEGER ip)
```

Get the value(s) of an integer attribute

vlistInqAttTxt

```
INTEGER FUNCTION vlistInqAttTxt(INTEGER vlistID, INTEGER varID,
                                CHARACTER*(*) name, INTEGER mlen,
                                CHARACTER*(*) tp)
```

Get the value(s) of a text attribute

vlistInqNatts

```
INTEGER FUNCTION vlistInqNatts(INTEGER vlistID, INTEGER varID, INTEGER nattsp)
```

Get number of variable attributes

vlistInqTaxis

```
INTEGER FUNCTION vlistInqTaxis(INTEGER vlistID)
```

Get the time axis

vlistInqVarCode

```
INTEGER FUNCTION vlistInqVarCode(INTEGER vlistID, INTEGER varID)
```

Get the Code number of a Variable

vlistInqVarDatatype

```
INTEGER FUNCTION vlistInqVarDatatype(INTEGER vlistID, INTEGER varID)
```

Get the data type of a Variable

vlistInqVarGrid

```
INTEGER FUNCTION vlistInqVarGrid(INTEGER vlistID, INTEGER varID)
```

Get the Grid ID of a Variable

vlistInqVarLongname

```
SUBROUTINE vlistInqVarLongname(INTEGER vlistID, INTEGER varID,
                               CHARACTER*(*) longname)
```

Get the longname of a Variable

vlistInqVarMissval

```
REAL*8 FUNCTION vlistInqVarMissval(INTEGER vlistID, INTEGER varID)
```

Get the missing value of a Variable

vlistInqVarName

```
SUBROUTINE vlistInqVarName(INTEGER vlistID, INTEGER varID, CHARACTER*(*) name)
```

Get the name of a Variable

vlistInqVarStdname

```
SUBROUTINE vlistInqVarStdname(INTEGER vlistID, INTEGER varID,
                              CHARACTER*(*) stdname)
```

Get the standard name of a Variable

vlistInqVarUnits

```
SUBROUTINE vlistInqVarUnits(INTEGER vlistID, INTEGER varID, CHARACTER*(*) units)
```

Get the units of a Variable

vlistInqVarZaxis

```
INTEGER FUNCTION vlistInqVarZaxis(INTEGER vlistID, INTEGER varID)
```

Get the Zaxis ID of a Variable

vlistNgrids

```
INTEGER FUNCTION vlistNgrids(INTEGER vlistID)
```

Number of grids in a variable list

vlistNvars

```
INTEGER FUNCTION vlistNvars(INTEGER vlistID)
```

Number of variables in a variable list

vlistNzaxis

```
INTEGER FUNCTION vlistNzaxis(INTEGER vlistID)
```

Number of zaxis in a variable list

zaxisCreate

```
INTEGER FUNCTION zaxisCreate(INTEGER zaxistype, INTEGER size)
```

Create a vertical Z-axis

zaxisDefLevels

```
SUBROUTINE zaxisDefLevels(INTEGER zaxisID, REAL*8 levels)
```

Define the levels of a Z-axis

zaxisDefLongname

```
SUBROUTINE zaxisDefLongname(INTEGER zaxisID, CHARACTER*(*) longname)
```

Define the longname of a Z-axis

zaxisDefName

```
SUBROUTINE zaxisDefName(INTEGER zaxisID, CHARACTER*(*) name)
```

Define the name of a Z-axis

zaxisDefUnits

```
SUBROUTINE zaxisDefUnits(INTEGER zaxisID, CHARACTER*(*) units)
```

Define the units of a Z-axis

zaxisDestroy

```
SUBROUTINE zaxisDestroy(INTEGER zaxisID)
```

Destroy a vertical Z-axis

zaxisInqLevel

```
REAL*8 FUNCTION zaxisInqLevel(INTEGER zaxisID, INTEGER levelID)
```

Get one level of a Z-axis

zaxisInqLevels

```
SUBROUTINE zaxisInqLevels(INTEGER zaxisID, REAL*8 levels)
```

Get all levels of a Z-axis

zaxisInqLongname

```
SUBROUTINE zaxisInqLongname(INTEGER zaxisID, CHARACTER*(*) longname)
```

Get the longname of a Z-axis

zaxisInqName

```
SUBROUTINE zaxisInqName(INTEGER zaxisID, CHARACTER*(*) name)
```

Get the name of a Z-axis

zaxisInqSize

```
INTEGER FUNCTION zaxisInqSize(INTEGER zaxisID)
```

Get the size of a Z-axis

zaxisInqType

```
INTEGER FUNCTION zaxisInqType(INTEGER zaxisID)
```

Get the type of a Z-axis

zaxisInqUnits

```
SUBROUTINE zaxisInqUnits(INTEGER zaxisID, CHARACTER*(*) units)
```

Get the units of a Z-axis

B. Examples

This appendix contains complete examples to write, read and copy a dataset with the **CDI** library.

B.1. Write a dataset

Here is an example using **CDI** to write a netCDF dataset with 2 variables on 3 time steps. The first variable is a 2D field on surface level and the second variable is a 3D field on 5 pressure levels. Both variables are on the same lon/lat grid.

```
      PROGRAM CDIWRITE

      IMPLICIT NONE

      INCLUDE 'cdi.inc'

      INTEGER nlon, nlat, nlev, nts
      PARAMETER (nlon = 12) ! Number of longitudes
      PARAMETER (nlat = 6) ! Number of latitudes
      PARAMETER (nlev = 5) ! Number of levels
      PARAMETER (nts = 3) ! Number of time steps

      INTEGER gridID, zaxisID1, zaxisID2, taxisID
      INTEGER vlistID, varID1, varID2, streamID, tsID
      INTEGER i, nmiss, status
      REAL*8 lons(nlon), lats(nlat), levs(nlev)
      REAL*8 var1(nlon*nlat), var2(nlon*nlat*nlev)

      DATA lons /0, 30, 60, 90, 120, 150, 180, 210, 240, 270, 300, 330/
      DATA lats /-75, -45, -15, 15, 45, 75/
      DATA levs /101300, 92500, 85000, 50000, 20000/

      nmiss = 0

!     Create a regular lon/lat grid
      gridID = gridCreate(GRID_LONLAT, nlon*nlat)
      CALL gridDefXsize(gridID, nlon)
      CALL gridDefYsize(gridID, nlat)
      CALL gridDefXvals(gridID, lons)
      CALL gridDefYvals(gridID, lats)

!     Create a surface level Z-axis
      zaxisID1 = zaxisCreate(ZAXIS_SURFACE, 1)

!     Create a pressure level Z-axis
      zaxisID2 = zaxisCreate(ZAXIS_PRESSURE, nlev)
      CALL zaxisDefLevels(zaxisID2, levs)

!     Create a variable list
      vlistID = vlistCreate()
```

```
!       Define the variables
        varID1 = vlistDefVar(vlistID, gridID, zaxisID1, TIME_VARIABLE)
        varID2 = vlistDefVar(vlistID, gridID, zaxisID2, TIME_VARIABLE)

!       Define the variable names
        CALL vlistDefVarName(vlistID, varID1, "varname1")
        CALL vlistDefVarName(vlistID, varID2, "varname2")

!       Create a Time axis
        taxisID = taxisCreate(TAXIS_ABSOLUTE)

!       Assign the Time axis to the variable list
        CALL vlistDefTaxis(vlistID, taxisID)

!       Create a dataset in netCDF format
        streamID = streamOpenWrite("example.nc", FILETYPE_NC)
        IF ( streamID < 0 ) THEN
           WRITE(0,*) cdiStringError(streamID)
           STOP
        END IF

!       Assign the variable list to the dataset
        CALL streamDefVlist(streamID, vlistID)

!       Loop over the number of time steps
        DO tsID = 0, nts-1
!          Set the verification date to 1985-01-01 + tsID
           CALL taxisDefVdate(taxisID, 19850101+tsID)
!          Set the verification time to 12:00:00
           CALL taxisDefVtime(taxisID, 120000)
!          Define the time step
           status = streamDefTimestep(streamID, tsID)

!          Init var1 and var2
           DO i = 1, nlon*nlat
              var1(i) = 1.1
           END DO
           DO i = 1, nlon*nlat*nlev
              var2(i) = 2.2
           END DO

!          Write var1 and var2
           CALL streamWriteVar(streamID, varID1, var1, nmiss)
           CALL streamWriteVar(streamID, varID2, var2, nmiss)
        END DO

!       Close the output stream
        CALL streamClose(streamID)

!       Destroy the objects
        CALL vlistDestroy(vlistID)
        CALL taxisDestroy(taxisID)
        CALL zaxisDestroy(zaxisID1)
        CALL zaxisDestroy(zaxisID2)
        CALL gridDestroy(gridID)

        END
```

B.1.1. Result

This is the `ncdump -h` output of the resulting netCDF file `example.nc`.

```
netcdf example {
dimensions:
        lon = 12 ;
        lat = 6 ;
        lev = 5 ;
        time = UNLIMITED ; // (3 currently)
variables:
        double lon(lon) ;
                lon:long_name = "longitude" ;
                lon:units = "degrees_east" ;
                lon:standard_name = "longitude" ;
        double lat(lat) ;
                lat:long_name = "latitude" ;
                lat:units = "degrees_north" ;
                lat:standard_name = "latitude" ;
        double lev(lev) ;
                lev:long_name = "pressure" ;
                lev:units = "Pa" ;
        double time(time) ;
                time:units = "day as %Y%m%d.%f" ;
        float varname1(time, lat, lon) ;
        float varname2(time, lev, lat, lon) ;
data:

 lon = 0, 30, 60, 90, 120, 150, 180, 210, 240, 270, 300, 330 ;

 lat = -75, -45, -15, 15, 45, 75 ;

 lev = 101300, 92500, 85000, 50000, 20000 ;

 time = 19850101.5, 19850102.5, 19850103.5 ;
}
```

B.2. Read a dataset

This example reads the netCDF file `example.nc` from Appendix B.1.

```
      PROGRAM CDIREAD

      IMPLICIT NONE

      INCLUDE 'cdi.inc'

      INTEGER nlon, nlat, nlev, nts
      PARAMETER (nlon = 12) ! Number of longitudes
      PARAMETER (nlat = 6) ! Number of latitudes
      PARAMETER (nlev = 5) ! Number of levels
      PARAMETER (nts = 3) ! Number of time steps

      INTEGER gridID, zaxisID1, zaxisID2, taxisID
      INTEGER vlistID, varID1, varID2, streamID, tsID
      INTEGER nmiss, status, vdate, vtime
      REAL*8 var1(nlon*nlat), var2(nlon*nlat*nlev)
```

```
!       Open the dataset
        streamID = streamOpenRead("example.nc")
        IF ( streamID < 0 ) THEN
           WRITE(0,*) cdiStringError(streamID)
           STOP
        END IF

!       Get the variable  list  of the  dataset
        vlistID  = streamInqVlist(streamID)

!       Set the  variable  IDs
        varID1 = 0
        varID2 = 1

!       Get the Time axis from the variable  list
        taxisID  = vlistInqTaxis( vlistID )

!       Loop over the number of time steps
        DO tsID = 0, nts-1
!          Inquire  the  time  step
           status = streamInqTimestep(streamID, tsID)

!          Get the  verification  date and time
           vdate = taxisInqVdate(taxisID)
           vtime = taxisInqVtime(taxisID)
           WRITE(0, *)"read_timestep_",tsID+1,"date=",vdate,"time=",vtime


!          Read var1 and var2
           CALL streamReadVar(streamID, varID1, var1, nmiss)
           CALL streamReadVar(streamID, varID2, var2, nmiss)
        END DO

!       Close the  input  stream
        CALL streamClose(streamID)

        END
```

B.3. Copy a dataset

This example reads the netCDF file `example.nc` from Appendix B.1 and writes the result to a GRIB dataset by simple setting the output file type to `FILETYPE_GRB`.

```
        PROGRAM CDICOPY

        IMPLICIT NONE

        INCLUDE 'cdi.inc'

        INTEGER nlon, nlat, nlev, nts
        PARAMETER (nlon = 12) ! Number of longitudes
        PARAMETER (nlat = 6) ! Number of latitudes
        PARAMETER (nlev = 5) ! Number of levels
        PARAMETER (nts = 3) ! Number of time steps
```

```
      INTEGER gridID, zaxisID1, zaxisID2, tsID
      INTEGER vlistID1, vlistID2, varID1, varID2, streamID1, streamID2
      INTEGER i, nmiss, status
      REAL*8 var1(nlon*nlat), var2(nlon*nlat*nlev)

!     Open the input dataset
      streamID1 = streamOpenRead("example.nc")
      IF ( streamID1 < 0 ) THEN
         WRITE(0,*) cdiStringError(streamID1)
         STOP
      END IF

!     Get the variable list of the dataset
      vlistID1 = streamInqVlist(streamID1)

!     Set the variable IDs
      varID1 = 0
      varID2 = 1

!     Open the output dataset (GRIB format)
      streamID2 = streamOpenWrite("example.grb", FILETYPE_GRB)
      IF ( streamID2 < 0 ) THEN
         WRITE(0,*) cdiStringError(streamID2)
         STOP
      END IF

      vlistID2 = vlistDuplicate(vlistID1)

      CALL streamDefVlist(streamID2, vlistID2)

!     Loop over the number of time steps
      DO tsID = 0, nts-1
!        Inquire the input time step */
         status = streamInqTimestep(streamID1, tsID)

!        Define the output time step
         status = streamDefTimestep(streamID2, tsID)

!        Read var1 and var2
         CALL streamReadVar(streamID1, varID1, var1, nmiss)
         CALL streamReadVar(streamID1, varID2, var2, nmiss)

!        Write var1 and var2
         CALL streamWriteVar(streamID2, varID1, var1, nmiss)
         CALL streamWriteVar(streamID2, varID2, var2, nmiss)
      END DO

!     Close the streams
      CALL streamClose(streamID1)
      CALL streamClose(streamID2)

      END
```

B.4. Fortran 2003: mo_cdi and iso_c_binding

This is the Fortran 2003 version of the reading and writing examples above. The main differenc to `cfortran.h` is the character handling. Here `CHARACTER(type=c_char)` is used instead of `CHARACTER`. Additionally plain fortran charcters and character variables have to be convertet to C charcters by

- appending `'\0'` with `//C_NULL_CHAR`
- prepending `C_CHAR_` to plain charcters
- take `ctrim` from `mo_cdi` for `CHARACTER(type=c_char)` variables

```fortran
PROGRAM CDIREADF2003
  use iso_c_binding
  use mo_cdi

  IMPLICIT NONE

  INTEGER :: gsize, nlevel, nvars, code
  INTEGER :: vdate, vtime, nmiss, status, ilev
  INTEGER :: streamID, varID, gridID, zaxisID
  INTEGER :: tsID, vlistID, taxisID
  DOUBLE PRECISION, ALLOCATABLE :: field(:,:)
  CHARACTER(kind=c_char), POINTER, DIMENSION(:) :: &
       msg, cdi_version
  CHARACTER(kind=c_char, LEN = cdi_max_name + 1) :: &
       name, longname, units
  INTEGER :: name_c_len, longname_c_len, units_c_len

  cdi_version => cdiLibraryVersion()

  WRITE (0, '(a,132a)') 'cdi_version: ', cdi_version

  ! Open the dataset
  streamID = streamOpenRead(C_CHAR_"example.nc"//C_NULL_CHAR)
  IF ( streamID < 0 ) THEN
    PRINT *, 'Could not Read the file.'
    msg => cdiStringError(streamID)
    WRITE(0,'(132a)') msg
    STOP 1
  END IF

  ! Get the variable list of the dataset
  vlistID = streamInqVlist(streamID)

  nvars = vlistNvars(vlistID)

  DO varID = 0, nvars-1
    code = vlistInqVarCode(vlistID, varID)
    CALL vlistInqVarName(vlistID, varID, name)
    CALL vlistInqVarLongname(vlistID, varID, longname)
    CALL vlistInqVarUnits(vlistID, varID, units)

    ! CALL ctrim(name)
    ! CALL ctrim(longname)
    ! CALL ctrim(units)
```

```fortran
    longname_c_len = c_len(longname)
    name_c_len = c_len(name)
    units_c_len  = c_len(units)
    PRINT '(a,2(i0,a),132a)', 'Parameter: ', varID+1, ' ', code, ' ', &
         name(1:name_c_len), ' ', longname(1:longname_c_len), ' ', &
         units(1:units_c_len), ' |'

  END DO

  ! Get the Time axis form the variable list
  taxisID = vlistInqTaxis(vlistID)

  ! Loop over the time steps
  DO tsID = 0, 999999
    ! Read the time step
    status = streamInqTimestep(streamID, tsID)
    IF ( status == 0 ) exit

    ! Get the verification date and time
    vdate = taxisInqVdate(taxisID)
    vtime = taxisInqVtime(taxisID)

    PRINT '(a,i3,i10,i10)', 'Timestep: ', tsID+1, vdate, vtime

    ! Read the variables at the current timestep
    DO varID = 0, nvars-1
      gridID = vlistInqVarGrid(vlistID, varID)
      gsize  = gridInqSize(gridID)
      zaxisID = vlistInqVarZaxis(vlistID, varID)
      nlevel  = zaxisInqSize(zaxisID)
      ALLOCATE(field(gsize, nlevel))
      CALL streamReadVar(streamID, varID, field, nmiss)
      DO ilev = 1, nlevel
        PRINT '(a,i3,a,i3,a,f10.5,1x,f10.5)', '   var=', varID+1, &
             ' level=', ilev, ':', &
             MINVAL(field(:,ilev)), MAXVAL(field(:,ilev))
      END DO
      DEALLOCATE(field)
    END DO
  END DO

  ! Close the input stream
  CALL streamClose(streamID)

END PROGRAM CDIREADF2003
```

```fortran
      PROGRAM CDIWRITEF2003

      USE iso_c_binding
      USE mo_cdi

      IMPLICIT NONE

      INTEGER nlon, nlat, nlev, nts
      PARAMETER (nlon = 12) ! Number of longitudes
      PARAMETER (nlat = 6) ! Number of latitudes
```

```
      PARAMETER (nlev = 5) ! Number of levels
      PARAMETER (nts = 3) ! Number of time steps

      INTEGER gridID, zaxisID1, zaxisID2, taxisID
      INTEGER vlistID, varID1, varID2, streamID, tsID
      INTEGER i, nmiss, status
      DOUBLE PRECISION lons(nlon), lats(nlat), levs(nlev)
      DOUBLE PRECISION var1(nlon*nlat), var2(nlon*nlat*nlev)
      CHARACTER(len=256, kind=c_char) :: varname
      CHARACTER(kind=c_char,len=1), POINTER :: msg(:)

      DATA lons /0, 30, 60, 90, 120, 150, 180, 210, 240, 270, 300, 330/
      DATA lats /-75, -45, -15, 15, 45, 75/
      DATA levs /101300, 92500, 85000, 50000, 20000/

      nmiss = 0

!     Create a regular lon/lat grid
      gridID = gridCreate(GRID_LONLAT, nlon*nlat)
      CALL gridDefXsize(gridID, nlon)
      CALL gridDefYsize(gridID, nlat)
      CALL gridDefXvals(gridID, lons)
      CALL gridDefYvals(gridID, lats)

!     Create a surface level Z-axis
      zaxisID1 = zaxisCreate(ZAXIS_SURFACE, 1)

!     Create a pressure level Z-axis
      zaxisID2 = zaxisCreate(ZAXIS_PRESSURE, nlev)
      CALL zaxisDefLevels(zaxisID2, levs)

!     Create a variable list
      vlistID = vlistCreate()

!     Define the variables
      varID1 = vlistDefVar(vlistID, gridID, zaxisID1, TIME_VARIABLE)
      varID2 = vlistDefVar(vlistID, gridID, zaxisID2, TIME_VARIABLE)

!     Define the variable names
      varname = "varname1" // c_null_char
      CALL vlistDefVarName(vlistID, varID1, varname)
      CALL vlistDefVarName(vlistID, varID2, C_CHAR_"varname2"//C_NULL_CHAR)

!     Create a Time axis
      taxisID = taxisCreate(TAXIS_ABSOLUTE)

!     Assign the Time axis to the variable list
      CALL vlistDefTaxis(vlistID, taxisID)

!     Create a dataset in netCDF format
      streamID = streamOpenWrite(C_CHAR_"example.nc"//C_NULL_CHAR, FILETYPE_NC)
      IF ( streamID < 0 ) THEN
         msg => cdiStringError(streamID)
         WRITE(0,'(132a)') msg
         STOP 1
      END IF
```

```
!      Assign the variable list to the dataset
       CALL streamDefVlist(streamID, vlistID)

!      Loop over the number of time steps
       DO tsID = 0, nts-1
!         Set the verification date to 1985-01-01 + tsID
          CALL taxisDefVdate(taxisID, 19850101+tsID)
!         Set the verification time to 12:00:00
          CALL taxisDefVtime(taxisID, 120000)
!         Define the time step
          status = streamDefTimestep(streamID, tsID)

!         Init var1 and var2
          DO i = 1, nlon*nlat
             var1(i) = 1.1
          END DO
          DO i = 1, nlon*nlat*nlev
             var2(i) = 2.2
          END DO

!         Write var1 and var2
          CALL streamWriteVar(streamID, varID1, var1, nmiss)
          CALL streamWriteVar(streamID, varID2, var2, nmiss)
       END DO

!      Close the output stream
       CALL streamClose(streamID)

!      Destroy the objects
       CALL vlistDestroy(vlistID)
       CALL taxisDestroy(taxisID)
       CALL zaxisDestroy(zaxisID1)
       CALL zaxisDestroy(zaxisID2)
       CALL gridDestroy(gridID)

       END PROGRAM CDIWRITEF2003
```

C. Environment Variables

The following table describes the environment variables that affect **CDI**.

Variable name	Default	Description
CDI_INVENTORY_MODE	None	Set to time to skip double variable entries.
CDI_VERSION_INFO	1	Set to 0 to disable netCDF global attribute CDI.

Function index

Z