CDI C Manual

Climate Data Interface
Version 1.4.7
January 2011

Uwe Schulzweida
Max-Planck-Institute for Meteorology

Contents

1. Introduction

CDI is an Interface to access Climate 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 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 library should be installed:

- Unidata netCDF library (`http://www.unidata.ucar.edu/packages/netcdf/index.html`) version 3 or higher. This is needed to read/write netCDF files 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 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 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 edition 1

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:

Grid type

0	Latitude/longitude grid
3	Lambert conformal grid
4	Gaussian latitude/longitude grid
10	Rotated latitude/longitude grid
50	Spherical Harmonic Coefficients
192	Icosahedral-hexagonal GME grid

Level type

1	Surface level
100	Isobaric level
103	Altitude above mean sea level
105	Height above ground
107	Sigma level
109	Hybrid level
110	Layer between two hybrid levels
111	Depth below land surface
112	Layer between two depths below land surface
113	Isentropic (theta) level
160	Depth below sea level

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 (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. 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 HHMM
`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**)

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. 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

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.

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

```
#include "cdi.h"
```

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

```
cc -c -I/usr/local/cdi/include myprogram.c
```

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:

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

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

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

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

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

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_NC`	File type netCDF
`FILETYPE_NC2`	File type netCDF version 2 (64-bit)
`FILETYPE_NC4`	File type netCDF-4 classic (HDF5)
`FILETYPE_SRV`	File type SERVICE
`FILETYPE_EXT`	File type EXTRA
`FILETYPE_IEG`	File type IEG

NetCDF is 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

```
int streamOpenWrite(const char *path, int 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_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"
   ...
int streamID;
   ...
streamID = streamOpenWrite("foo.nc", FILETYPE_NC);
if ( streamID < 0 ) handle_error(streamID);
   ...
```

4.1.2. Open a dataset for reading: `streamOpenRead`

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

Usage

```
int streamOpenRead(const char *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"
   ...
int streamID;
   ...
streamID = streamOpenRead("foo.nc");
if ( streamID < 0 ) handle_error(streamID);
   ...
```

4.1.3. Close an open dataset: `streamClose`

The function `streamClose` closes an open dataset.

Usage

```
void streamClose(int 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

```
int streamInqFiletype(int 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_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

```
void streamDefByteorder(int streamID, int byteorder);
```

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

```
int streamInqByteorder(int 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

```
void streamDefVlist(int streamID, int vlistID);
```

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

```
int streamInqVlist(int 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

```
int streamDefTimestep(int streamID, int tsID);
```

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

```
int streamInqTimestep(int streamID, int 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.

Usage

```
void streamWriteVar(int streamID, int varID, const double *data, int nmiss);
```

`streamID`	Stream ID, from a previous call to `streamOpenRead` or `streamOpenWrite`
`varID`	Variable identifier
`data`	Pointer to a block of data values to be written
`nmiss`	Number of missing values

4.1.12. Read a variable: `streamReadVar`

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

Usage

```
void streamReadVar(int streamID, int varID, double *data, int *nmiss);
```

`streamID`	Stream ID, from a previous call to `streamOpenRead` or `streamOpenWrite`
`varID`	Variable identifier
`data`	Pointer to the location into which the data value is read
`nmiss`	Number of missing values

4.1.13. 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.

Usage

```
void streamWriteVarSlice(int streamID, int varID, int levelID, const double *data,
                         int nmiss);
```

`streamID`	Stream ID, from a previous call to `streamOpenRead` or `streamOpenWrite`
`varID`	Variable identifier
`levelID`	Level identifier
`data`	Pointer to a block of data values to be written
`nmiss`	Number of missing values

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

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

Usage

```
void streamReadVarSlice(int streamID, int varID, int levelID, double *data,
                        int *nmiss);
```

`streamID`	Stream ID, from a previous call to `streamOpenRead` or `streamOpenWrite`
`varID`	Variable identifier
`levelID`	Level identifier
`data`	Pointer to the location into which the data value is read
`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

```
int vlistCreate(void);
```

Example

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

```
#include "cdi.h"
   ...
int vlistID, varID;
   ...
vlistID = vlistCreate();
varID = vlistDefVar(vlistID, gridID, zaxisID, TIME_VARIABLE);
   ...
streamDefVlist(streamID, vlistID);
   ...
vlistDestroy(vlistID);
   ...
```

4.2.2. Destroy a variable list: `vlistDestroy`

Usage

```
void vlistDestroy(int 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

```
void vlistCopy(int vlistID2, int 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

```
int vlistDuplicate(int vlistID);
```

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

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

```
void vlistCat(int vlistID2, int 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

```
void vlistCopyFlag(int vlistID2, int 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

```
int vlistNvars(int vlistID);
```

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

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

```
int vlistNgrids(int vlistID);
```

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

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

```
int vlistNzaxis(int vlistID);
```

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

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

```
void vlistDefTaxis(int vlistID, int 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

```
int vlistInqTaxis(int vlistID);
```

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

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:

`TIME_CONSTANT`	For time constant variables
`TIME_VARIABLE`	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

4.3.1. Define a Variable: `vlistDefVar`

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

Usage

```
int vlistDefVar(int vlistID, int gridID, int zaxisID, int timeID);
```

`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`
`timeID`	One of the set of predefined **CDI** time identifiers. The valid **CDI** time identifiers are `TIME_CONSTANT` and `TIME_VARIABLE`.

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"
   ...
int vlistID, varID;
   ...
vlistID = vlistCreate();
varID = vlistDefVar(vlistID, gridID, zaxisID, TIME_VARIABLE);
   ...
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

```
int vlistInqVarGrid(int vlistID, int varID);
```

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

```
int vlistInqVarZaxis(int vlistID, int varID);
```

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

```
void vlistDefVarCode(int vlistID, int varID, int 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

```
int vlistInqVarCode(int vlistID, int varID);
```

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

```
void vlistDefVarName(int vlistID, int varID, const char *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

```
void vlistInqVarName(int vlistID, int varID, char *name);
```

`vlistID`	Variable list ID, from a previous call to `vlistCreate`
`varID`	Variable identifier
`name`	Variable name

Result

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

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

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

Usage

```
void vlistDefVarLongname(int vlistID, int varID, const char *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.

Usage

```
void vlistInqVarLongname(int vlistID, int varID, char *longname);
```

`vlistID`	Variable list ID, from a previous call to `vlistCreate`
`varID`	Variable identifier
`longname`	Variable description

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

```
void vlistDefVarStdname(int vlistID, int varID, const char *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.

Usage

```
void vlistInqVarStdname(int vlistID, int varID, char *stdname);
```

`vlistID`	Variable list ID, from a previous call to `vlistCreate`
`varID`	Variable identifier
`stdname`	Variable standard 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

```
void vlistDefVarUnits(int vlistID, int varID, const char *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.

Usage

```
void vlistInqVarUnits(int vlistID, int varID, char *units);
```

`vlistID`	Variable list ID, from a previous call to `vlistCreate`
`varID`	Variable identifier
`units`	Variable units

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

```
void vlistDefVarDatatype(int vlistID, int varID, int 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` and `DATATYPE_FLT64`.

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

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

Usage

```
int vlistInqVarDatatype(int vlistID, int varID);
```

`vlistID`	Variable list ID, from a previous call to `vlistCreate`
`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` and `DATATYPE_FLT64`.

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

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

Usage

```
void vlistDefVarMissval(int vlistID, int varID, double 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

```
double vlistInqVarMissval(int vlistID, int varID);
```

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

```
int vlistInqNatts(int vlistID, int varID, int *nattsp);
```

`vlistID`	Variable list ID, from a previous call to `vlistCreate`
`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 `vlistInqNatts` gets information about an attribute.

Usage

```
int vlistInqAtt(int vlistID, int varID, int attnum, char *name, int *typep, int *lenp);
```

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

```
    int vlistDefAttInt(int vlistID, int varID, const char *name, int type, int len,
                       const int *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

```
    int vlistInqAttInt(int vlistID, int varID, const char *name, int mlen, int *ip);
```

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

```
    int vlistDefAttFlt(int vlistID, int varID, const char *name, int type, int len,
                       const double *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

```
    int vlistInqAttFlt(int vlistID, int varID, const char *name, int mlen, int *dp);
```

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

```
int vlistDefAttTxt(int vlistID, int varID, const char *name, int len, const char *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

```
int vlistInqAttTxt(int vlistID, int varID, const char *name, int mlen, int *tp);
```

`vlistID`	Variable list ID, from a previous call to `vlistCreate`
`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_GAUSSIAN	Gaussian latitude/longitude grid
GRID_LONLAT	Equidistant longitude/latitude grid
GRID_LCC	Lambert conformal conic grid
GRID_SPECTRAL	Spherical harmonic coefficients
GRID_GME	Icosahedral-hexagonal GME grid
GRID_CURVILINEAR	Curvilinear grid
GRID_CELL	Unstructured grid cells

4.5.1. Create a horizontal Grid: `gridCreate`

The function `gridCreate` creates a horizontal Grid.

Usage

```
int gridCreate(int gridtype, int 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_CELL`.
`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
   ...
double lons[nlon] = {0, 30, 60, 90, 120, 150, 180, 210, 240, 270, 300, 330};
double lats[nlat]  = {-75, -45, -15, 15, 45, 75};
int gridID;
   ...
gridID = gridCreate(GRID_LONLAT, nlon*nlat);
gridDefXsize(gridID, nlon);
gridDefYsize(gridID, nlat);
gridDefXvals(gridID, lons);
gridDefYvals(gridID, lats);
   ...
```

4.5.2. Destroy a horizontal Grid: `gridDestroy`

Usage

```
void gridDestroy(int 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

```
int gridDuplicate(int gridID);
```

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

```
int gridInqType(int gridID);
```

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

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_CELL`.

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

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

Usage

```
int gridInqSize(int gridID);
```

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

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

```
void gridDefXsize(int gridID, int 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

```
void gridInqXsize(int gridID);
```

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

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

```
void gridDefYsize(int gridID, int 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

```
void gridInqYsize(int gridID);
```

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

Result

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

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

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

Usage

```
void gridDefXvals(int gridID, const double *xvals);
```

`gridID`	Grid ID, from a previous call to `gridCreate`
`xvals`	X-values of the grid

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

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

Usage

```
int gridInqXvals(int gridID, double *xvals);
```

`gridID`	Grid ID, from a previous call to `gridCreate`
`xvals`	X-values of the grid

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.

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

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

Usage

```
void gridDefYvals(int gridID, const double *yvals);
```

`gridID`	Grid ID, from a previous call to `gridCreate`
`yvals`	Y-values of the grid

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

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

Usage

```
int gridInqYvals(int gridID, double *yvals);
```

`gridID`	Grid ID, from a previous call to `gridCreate`
`yvals`	Y-values of the grid

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.

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

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

Usage

```
void gridDefXbounds(int gridID, const double *xbounds);
```

`gridID`	Grid ID, from a previous call to `gridCreate`
`xbounds`	X-bounds of the grid

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

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

Usage

```
int gridInqXbounds(int gridID, double *xbounds);
```

`gridID`	Grid ID, from a previous call to `gridCreate`
`xbounds`	X-bounds of the grid

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.

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

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

Usage

```
void gridDefYbounds(int gridID, const double *ybounds);
```

`gridID`	Grid ID, from a previous call to `gridCreate`
`ybounds`	Y-bounds of the grid

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

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

Usage

```
int gridInqYbounds(int gridID, double *ybounds);
```

`gridID`	Grid ID, from a previous call to `gridCreate`
`ybounds`	Y-bounds of the grid

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.

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

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

Usage

```
void gridDefXname(int gridID, const char *name);
```

`gridID`	Grid ID, from a previous call to `gridCreate`
`name`	Name of the X-axis

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

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

Usage

```
void gridInqXname(int gridID, const char *name);
```

`gridID`	Grid ID, from a previous call to `gridCreate`
`name`	Name of the X-axis

Result

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

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

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

Usage

```
void gridDefXlongname(int gridID, const char *longname);
```

`gridID`	Grid ID, from a previous call to `gridCreate`
`longname`	Longname of the X-axis

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

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

Usage

```
void gridInqXlongname(int gridID, const char *longname);
```

`gridID`	Grid ID, from a previous call to `gridCreate`
`longname`	Longname of the X-axis

Result

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

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

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

Usage

```
void gridDefXunits(int gridID, const char *units);
```

`gridID`	Grid ID, from a previous call to `gridCreate`
`units`	Units of the X-axis

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

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

Usage

```
void gridInqXunits(int gridID, const char *units);
```

`gridID`	Grid ID, from a previous call to `gridCreate`
`units`	Units of the X-axis

Result

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

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

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

Usage

```
    void gridDefYname(int gridID, const char *name);
```

`gridID`	Grid ID, from a previous call to `gridCreate`
`name`	Name of the Y-axis

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

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

Usage

```
    void gridInqYname(int gridID, const char *name);
```

`gridID`	Grid ID, from a previous call to `gridCreate`
`name`	Name of the Y-axis

Result

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

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

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

Usage

```
    void gridDefYlongname(int gridID, const char *longname);
```

`gridID`	Grid ID, from a previous call to `gridCreate`
`longname`	Longname of the Y-axis

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

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

Usage

```
    void gridInqXlongname(int gridID, const char *longname);
```

`gridID`	Grid ID, from a previous call to `gridCreate`
`longname`	Longname of the Y-axis

Result

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

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

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

Usage

```
void gridDefYunits(int gridID, const char *units);
```

`gridID`	Grid ID, from a previous call to `gridCreate`
`units`	Units of the Y-axis

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

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

Usage

```
void gridInqYunits(int gridID, const char *units);
```

`gridID`	Grid ID, from a previous call to `gridCreate`
`units`	Units of the Y-axis

Result

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

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_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_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

```
int zaxisCreate(int zaxistype, int 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_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
   ...
double levs[nlev] = {101300, 92500, 85000, 50000, 20000};
int zaxisID;
   ...
zaxisID = zaxisCreate(ZAXIS_PRESSURE, nlev);
zaxisDefLevels(zaxisID, levs);
   ...
```

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

Usage

```
void zaxisDestroy(int 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

```
int zaxisInqType(int zaxisID);
```

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

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_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

```
int zaxisInqSize(int zaxisID);
```

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

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

```
void zaxisDefLevels(int zaxisID, const double *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

```
void zaxisInqLevels(int zaxisID, double *levels);
```

`zaxisID`	Z-axis ID, from a previous call to `zaxisCreate`
`levels`	Levels of the Z-axis

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

```
double zaxisInqLevel(int zaxisID, int levelID);
```

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

```
void zaxisDefName(int zaxisID, const char *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

```
void zaxisInqName(int zaxisID, char *name);
```

`zaxisID`	Z-axis ID, from a previous call to `zaxisCreate`
`name`	Name of the Z-axis

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

```
void zaxisDefLongname(int zaxisID, const char *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

```
void zaxisInqLongname(int zaxisID, char *longname);
```

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

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

```
void zaxisDefUnits(int zaxisID, const char *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

```
void zaxisInqUnits(int zaxisID, char *units);
```

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

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. 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

```
int taxisCreate(int 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"
   ...
int taxisID;
   ...
taxisID = taxisCreate(TAXIS_RELATIVE);
taxisDefCalendar(taxisID, CALENDAR_STANDARD);
taxisDefRdate(taxisID, 19850101);
taxisDefRtime(taxisID, 120000);
   ...
```

4.7.2. Destroy a Time axis: `taxisDestroy`

Usage

```
void taxisDestroy(int taxisID);
```

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

4.7.3. Define the reference date: `taxisDefRdate`

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

Usage

```
void taxisDefRdate(int taxisID, int 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

```
int taxisInqRdate(int taxisID);
```

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

Result

`taxisInqVdate` returns the reference date.

4.7.5. Define the reference time: `taxisDefRtime`

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

Usage

```
void taxisDefRtime(int taxisID, int 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

```
int taxisInqRtime(int taxisID);
```

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

Result

`taxisInqVtime` returns the reference time.

4.7.7. Define the verification date: `taxisDefVdate`

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

Usage

```
void taxisDefVdate(int taxisID, int 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

```
int taxisInqVdate(int taxisID);
```

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

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

```
void taxisDefVtime(int taxisID, int 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

```
int taxisInqVtime(int taxisID);
```

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

Result

`taxisInqVtime` returns the verification time.

4.7.11. Define the calendar: `taxisDefCalendar`

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

Usage

```
void taxisDefCalendar(int taxisID, int 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

```
int taxisInqCalendar(int taxisID);
```

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

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_PROLEPRIC`, `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)

[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 C language bindings of the CDI library routines:

gridCreate

```
int gridCreate(int gridtype, int size);
```

Create a horizontal Grid

gridDefXbounds

```
void gridDefXbounds(int gridID, const double *xbounds);
```

Define the bounds of a X-axis

gridDefXlongname

```
void gridDefXlongname(int gridID, const char *longname);
```

Define the longname of a X-axis

gridDefXname

```
void gridDefXname(int gridID, const char *name);
```

Define the name of a X-axis

gridDefXsize

```
void gridDefXsize(int gridID, int xsize);
```

Define the number of values of a X-axis

gridDefXunits

```
void gridDefXunits(int gridID, const char *units);
```

Define the units of a X-axis

gridDefXvals

```
void gridDefXvals(int gridID, const double *xvals);
```

Define the values of a X-axis

gridDefYbounds

```
void gridDefYbounds(int gridID, const double *ybounds);
```

Define the bounds of a Y-axis

gridDefYlongname

```
void gridDefYlongname(int gridID, const char *longname);
```

Define the longname of a Y-axis

gridDefYname

```
void gridDefYname(int gridID, const char *name);
```

Define the name of a Y-axis

gridDefYsize

```
void gridDefYsize(int gridID, int ysize);
```

Define the number of values of a Y-axis

gridDefYunits

```
void gridDefYunits(int gridID, const char *units);
```

Define the units of a Y-axis

gridDefYvals

```
void gridDefYvals(int gridID, const double *yvals);
```

Define the values of a Y-axis

gridDestroy

```
void gridDestroy(int gridID);
```

Destroy a horizontal Grid

gridDuplicate

```
int gridDuplicate(int gridID);
```

Duplicate a horizontal Grid

gridInqSize

```
int gridInqSize(int gridID);
```

Get the size of a Grid

gridInqType

```
int gridInqType(int gridID);
```

Get the type of a Grid

gridInqXbounds

```
int gridInqXbounds(int gridID, double *xbounds);
```

Get the bounds of a X-axis

gridInqXlongname

```
void gridInqXlongname(int gridID, const char *longname);
```

Get the longname of a X-axis

gridInqXname

```
void gridInqXname(int gridID, const char *name);
```

Get the name of a X-axis

gridInqXsize

```
void gridInqXsize(int gridID);
```

Get the number of values of a X-axis

gridInqXunits

```
void gridInqXunits(int gridID, const char *units);
```

Get the units of a X-axis

gridInqXvals

```
int gridInqXvals(int gridID, double *xvals);
```

Get all values of a X-axis

gridInqYbounds

```
int gridInqYbounds(int gridID, double *ybounds);
```

Get the bounds of a Y-axis

gridInqYlongname

```
void gridInqXlongname(int gridID, const char *longname);
```

Get the longname of a Y-axis

gridInqYname

```
void gridInqYname(int gridID, const char *name);
```

Get the name of a Y-axis

gridInqYsize

```
void gridInqYsize(int gridID);
```

Get the number of values of a Y-axis

gridInqYunits

```
void gridInqYunits(int gridID, const char *units);
```

Get the units of a Y-axis

gridInqYvals

```
int gridInqYvals(int gridID, double *yvals);
```

Get all values of a Y-axis

streamClose

```
void streamClose(int streamID);
```

Close an open dataset

streamDefByteorder

```
void streamDefByteorder(int streamID, int byteorder);
```

Define the byte order

streamDefTimestep

```
int streamDefTimestep(int streamID, int tsID);
```

Define time step

streamDefVlist

```
void streamDefVlist(int streamID, int vlistID);
```

Define the variable list

streamInqByteorder

```
int streamInqByteorder(int streamID);
```

Get the byte order

streamInqFiletype

```
int streamInqFiletype(int streamID);
```

Get the filetype

streamInqTimestep

```
int streamInqTimestep(int streamID, int tsID);
```

Get time step

streamInqVlist

```
int streamInqVlist(int streamID);
```

Get the variable list

streamOpenRead

```
int streamOpenRead(const char *path);
```

Open a dataset for reading

streamOpenWrite

```
int streamOpenWrite(const char *path, int filetype);
```

Create a new dataset

streamReadVar

```
void streamReadVar(int streamID, int varID, double *data, int *nmiss);
```

Read a variable

streamReadVarSlice

```
void streamReadVarSlice(int streamID, int varID, int levelID, double *data,
                        int *nmiss);
```

Read a horizontal slice of a variable

streamWriteVar

```
void streamWriteVar(int streamID, int varID, const double *data, int nmiss);
```

Write a variable

streamWriteVarSlice

```
void streamWriteVarSlice(int streamID, int varID, int levelID, const double *data,
                         int nmiss);
```

Write a horizontal slice of a variable

taxisCreate

```
int taxisCreate(int taxistype);
```

Create a Time axis

taxisDefCalendar

```
void taxisDefCalendar(int taxisID, int calendar);
```

Define the calendar

taxisDefRdate

```
void taxisDefRdate(int taxisID, int rdate);
```

Define the reference date

taxisDefRtime

```
void taxisDefRtime(int taxisID, int rtime);
```

Define the reference time

taxisDefVdate

```
void taxisDefVdate(int taxisID, int vdate);
```

Define the verification date

taxisDefVtime

```
void taxisDefVtime(int taxisID, int vtime);
```

Define the verification time

taxisDestroy

```
void taxisDestroy(int taxisID);
```

Destroy a Time axis

taxisInqCalendar

```
int taxisInqCalendar(int taxisID);
```

Get the calendar

taxisInqRdate

```
int taxisInqRdate(int taxisID);
```

Get the reference date

taxisInqRtime

```
int taxisInqRtime(int taxisID);
```

Get the reference time

taxisInqVdate

```
int taxisInqVdate(int taxisID);
```

Get the verification date

taxisInqVtime

```
int taxisInqVtime(int taxisID);
```

Get the verification time

vlistCat

```
    void vlistCat(int vlistID2, int vlistID1);
```

Concatenate two variable lists

vlistCopy

```
    void vlistCopy(int vlistID2, int vlistID1);
```

Copy a variable list

vlistCopyFlag

```
    void vlistCopyFlag(int vlistID2, int vlistID1);
```

Copy some entries of a variable list

vlistCreate

```
    int vlistCreate(void);
```

Create a variable list

vlistDefAttFlt

```
    int vlistDefAttFlt(int vlistID, int varID, const char *name, int type, int len,
                       const double *dp);
```

Define a floating point attribute

vlistDefAttInt

```
    int vlistDefAttInt(int vlistID, int varID, const char *name, int type, int len,
                       const int *ip);
```

Define an integer attribute

vlistDefAttTxt

```
    int vlistDefAttTxt(int vlistID, int varID, const char *name, int len, const char *tp);
```

Define a text attribute

vlistDefTaxis

```
    void vlistDefTaxis(int vlistID, int taxisID);
```

Define the time axis

vlistDefVar

```
int vlistDefVar(int vlistID, int gridID, int zaxisID, int timeID);
```

Define a Variable

vlistDefVarCode

```
void vlistDefVarCode(int vlistID, int varID, int code);
```

Define the code number of a Variable

vlistDefVarDatatype

```
void vlistDefVarDatatype(int vlistID, int varID, int datatype);
```

Define the data type of a Variable

vlistDefVarLongname

```
void vlistDefVarLongname(int vlistID, int varID, const char *longname);
```

Define the long name of a Variable

vlistDefVarMissval

```
void vlistDefVarMissval(int vlistID, int varID, double missval);
```

Define the missing value of a Variable

vlistDefVarName

```
void vlistDefVarName(int vlistID, int varID, const char *name);
```

Define the name of a Variable

vlistDefVarStdname

```
void vlistDefVarStdname(int vlistID, int varID, const char *stdname);
```

Define the standard name of a Variable

vlistDefVarUnits

```
void vlistDefVarUnits(int vlistID, int varID, const char *units);
```

Define the units of a Variable

vlistDestroy

```
void vlistDestroy(int vlistID);
```

Destroy a variable list

vlistDuplicate

```
int vlistDuplicate(int vlistID);
```

Duplicate a variable list

vlistInqAtt

```
int vlistInqAtt(int vlistID, int varID, int attnum, char *name, int *typep, int *lenp);
```

Get information about an attribute

vlistInqAttFlt

```
int vlistInqAttFlt(int vlistID, int varID, const char *name, int mlen, int *dp);
```

Get the value(s) of a floating point attribute

vlistInqAttInt

```
int vlistInqAttInt(int vlistID, int varID, const char *name, int mlen, int *ip);
```

Get the value(s) of an integer attribute

vlistInqAttTxt

```
int vlistInqAttTxt(int vlistID, int varID, const char *name, int mlen, int *tp);
```

Get the value(s) of a text attribute

vlistInqNatts

```
int vlistInqNatts(int vlistID, int varID, int *nattsp);
```

Get number of variable attributes

vlistInqTaxis

```
int vlistInqTaxis(int vlistID);
```

Get the time axis

vlistInqVarCode

```
int vlistInqVarCode(int vlistID, int varID);
```

Get the Code number of a Variable

vlistInqVarDatatype

```
int vlistInqVarDatatype(int vlistID, int varID);
```

Get the data type of a Variable

vlistInqVarGrid

```
int vlistInqVarGrid(int vlistID, int varID);
```

Get the Grid ID of a Variable

vlistInqVarLongname

```
void vlistInqVarLongname(int vlistID, int varID, char *longname);
```

Get the longname of a Variable

vlistInqVarMissval

```
double vlistInqVarMissval(int vlistID, int varID);
```

Get the missing value of a Variable

vlistInqVarName

```
void vlistInqVarName(int vlistID, int varID, char *name);
```

Get the name of a Variable

vlistInqVarStdname

```
void vlistInqVarStdname(int vlistID, int varID, char *stdname);
```

Get the standard name of a Variable

vlistInqVarUnits

```
void vlistInqVarUnits(int vlistID, int varID, char *units);
```

Get the units of a Variable

vlistInqVarZaxis

```
int vlistInqVarZaxis(int vlistID, int varID);
```

Get the Zaxis ID of a Variable

vlistNgrids

```
int vlistNgrids(int vlistID);
```

Number of grids in a variable list

vlistNvars

```
int vlistNvars(int vlistID);
```

Number of variables in a variable list

vlistNzaxis

```
int vlistNzaxis(int vlistID);
```

Number of zaxis in a variable list

zaxisCreate

```
int zaxisCreate(int zaxistype, int size);
```

Create a vertical Z-axis

zaxisDefLevels

```
void zaxisDefLevels(int zaxisID, const double *levels);
```

Define the levels of a Z-axis

zaxisDefLongname

```
void zaxisDefLongname(int zaxisID, const char *longname);
```

Define the longname of a Z-axis

zaxisDefName

```
void zaxisDefName(int zaxisID, const char *name);
```

Define the name of a Z-axis

zaxisDefUnits

```
void zaxisDefUnits(int zaxisID, const char *units);
```

Define the units of a Z-axis

zaxisDestroy

```
void zaxisDestroy(int zaxisID);
```

Destroy a vertical Z-axis

zaxisInqLevel

```
double zaxisInqLevel(int zaxisID, int levelID);
```

Get one level of a Z-axis

zaxisInqLevels

```
void zaxisInqLevels(int zaxisID, double *levels);
```

Get all levels of a Z-axis

zaxisInqLongname

```
void zaxisInqLongname(int zaxisID, char *longname);
```

Get the longname of a Z-axis

zaxisInqName

```
void zaxisInqName(int zaxisID, char *name);
```

Get the name of a Z-axis

zaxisInqSize

```
int zaxisInqSize(int zaxisID);
```

Get the size of a Z-axis

zaxisInqType

```
int zaxisInqType(int zaxisID);
```

Get the type of a Z-axis

zaxisInqUnits

```
void zaxisInqUnits(int zaxisID, char *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.

```c
#include <stdio.h>
#include "cdi.h"

#define nlon   12 // Number of longitudes
#define nlat    6 // Number of latitudes
#define nlev    5 // Number of levels
#define nts     3 // Number of time steps

int main(void)
{
  int gridID, zaxisID1, zaxisID2, taxisID;
  int vlistID, varID1, varID2, streamID, tsID;
  int i, nmiss = 0;
  double lons[nlon] = {0, 30, 60, 90, 120, 150, 180, 210, 240, 270, 300, 330};
  double lats[nlat] = {-75, -45, -15, 15, 45, 75};
  double levs[nlev] = {101300, 92500, 85000, 50000, 20000};
  double var1[nlon*nlat];
  double var2[nlon*nlat*nlev];


  // Create a regular lon/lat grid
  gridID = gridCreate(GRID_LONLAT, nlon*nlat);
  gridDefXsize(gridID, nlon);
  gridDefYsize(gridID, nlat);
  gridDefXvals(gridID, lons);
  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);
  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
  vlistDefVarName(vlistID, varID1, "varname1");
  vlistDefVarName(vlistID, varID2, "varname2");

  // Create a Time axis
  taxisID = taxisCreate(TAXIS_ABSOLUTE);

  // Assign the Time axis to the variable list
  vlistDefTaxis(vlistID, taxisID);

  // Create a dataset in netCDF fromat
  streamID = streamOpenWrite("example.nc", FILETYPE_NC);
  if ( streamID < 0 )
    {
      fprintf(stderr, "%s\n", cdiStringError(streamID));
      return(1);
    }

  // Assign the variable list to the dataset
  streamDefVlist(streamID, vlistID);

  // Loop over the number of time steps
  for ( tsID = 0; tsID < nts; tsID++ )
    {
      // Set the verification date to 1985-01-01 + tsID
      taxisDefVdate(taxisID, 19850101+tsID);
      // Set the verification time to 12:00:00
      taxisDefVtime(taxisID, 120000);
      // Define the time step
      streamDefTimestep(streamID, tsID);

      // Init var1 and var2
      for ( i = 0; i < nlon*nlat;      i++ ) var1[i] = 1.1;
      for ( i = 0; i < nlon*nlat*nlev; i++ ) var2[i] = 2.2;

      // Write var1 and var2
      streamWriteVar(streamID, varID1, var1, nmiss);
      streamWriteVar(streamID, varID2, var2, nmiss);
    }

  // Close the output stream
  streamClose(streamID);

  // Destroy the objects
  vlistDestroy(vlistID);
  taxisDestroy(taxisID);
  zaxisDestroy(zaxisID1);
  zaxisDestroy(zaxisID2);
  gridDestroy(gridID);

  return 0;
}
```

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.

```
#include <stdio.h>
#include "cdi.h"

int nlon = 12; // Number of longitudes
int nlat =  6; // Number of latitudes
int nlev =  5; // Number of levels
int nts  =  3; // Number of time steps

int main(void)
{
  int taxisID, vlistID, varID1, varID2, streamID, tsID;
  int nmiss, vdate, vtime;
  double var1[nlon*nlat];
  double var2[nlon*nlat*nlev];


  // Open the dataset
  streamID = streamOpenRead("example.nc");
  if ( streamID < 0 )
```

```
    {
      fprintf (stderr, "%s\n", cdiStringError(streamID));
      return(1);
    }

  // 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
  for ( tsID = 0; tsID < nts; tsID++ )
    {
      // Inquire the time step
      streamInqTimestep(streamID, tsID);

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

      // Read var1 and var2
      streamReadVar(streamID, varID1, var1, &nmiss);
      streamReadVar(streamID, varID2, var2, &nmiss);
    }

  // Close the input stream
  streamClose(streamID);

  return 0;
}
```

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`.

```
#include <stdio.h>
#include "cdi.h"

int nlon = 12; // Number of longitudes
int nlat =  6; // Number of latitudes
int nlev =  5; // Number of levels
int nts  =  3; // Number of time steps

int main(void)
{
  int taxisID, vlistID1, vlistID2, varID1, varID2, streamID1, streamID2, tsID;
  int nmiss, vdate, vtime;
  double var1[nlon*nlat];
  double var2[nlon*nlat*nlev];

  // Open the input dataset
  streamID1 = streamOpenRead("example.nc");
  if ( streamID1 < 0 )
    {
      fprintf (stderr, "%s\n", cdiStringError(streamID1));
      return(1);
    }

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

  // Set the variable IDs
  varID1 = 0;
  varID2 = 1;

  // Get the Time axis from the variable list
  taxisID = vlistInqTaxis(vlistID1);

  // Open the output dataset (GRIB fromat)
  streamID2 = streamOpenWrite("example.grb", FILETYPE_GRB);
  if ( streamID2 < 0 )
    {
      fprintf (stderr, "%s\n", cdiStringError(streamID2));
      return(1);
    }

  vlistID2 = vlistDuplicate(vlistID1);

  streamDefVlist(streamID2, vlistID2);

  // Loop over the number of time steps
  for ( tsID = 0; tsID < nts; tsID++ )
    {
      // Inquire the input time step
      streamInqTimestep(streamID1, tsID);

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

      // Define the output time step
      streamDefTimestep(streamID2, tsID);

      // Read var1 and var2
      streamReadVar(streamID1, varID1, var1, &nmiss);
      streamReadVar(streamID1, varID2, var2, &nmiss);

      // Write var1 and var2
      streamWriteVar(streamID2, varID1, var1, nmiss);
      streamWriteVar(streamID2, varID2, var2, nmiss);
    }

  // Close the streams
  streamClose(streamID1);
  streamClose(streamID2);
  return 0;
}
```

Function index