

CDI C Manual

Climate Data Interface
Version 1.9.1
October 2017

Uwe Schulzweida
Max-Planck-Institute for Meteorology

Contents

1. Introduction	6
1.1. Building from sources	6
1.1.1. Compilation	6
1.1.2. Installation	7
2. File Formats	8
2.1. GRIB	8
2.1.1. GRIB edition 1	9
2.1.2. GRIB edition 2	9
2.2. NetCDF	9
2.3. SERVICE	10
2.4. EXTRA	10
2.5. IEG	11
3. Use of the CDI Library	12
3.1. Creating a dataset	12
3.2. Reading a dataset	12
3.3. Compiling and Linking with the CDI library	13
4. CDI modules	14
4.1. Dataset functions	14
4.1.1. Create a new dataset: <code>streamOpenWrite</code>	14
4.1.2. Open a dataset for reading: <code>streamOpenRead</code>	15
4.1.3. Close an open dataset: <code>streamClose</code>	16
4.1.4. Get the filetype: <code>streamInqFiletype</code>	16
4.1.5. Define the byte order: <code>streamDefByteorder</code>	16
4.1.6. Get the byte order: <code>streamInqByteorder</code>	16
4.1.7. Define the variable list: <code>streamDefVlist</code>	17
4.1.8. Get the variable list: <code>streamInqVlist</code>	17
4.1.9. Define time step: <code>streamDefTimestep</code>	17
4.1.10. Get time step: <code>streamInqTimestep</code>	17
4.1.11. Write a variable: <code>streamWriteVar</code>	18
4.1.12. Write a variable: <code>streamWriteVarF</code>	18
4.1.13. Write a horizontal slice of a variable: <code>streamWriteVarSlice</code>	18
4.1.14. Write a horizontal slice of a variable: <code>streamWriteVarSliceF</code>	18
4.1.15. Read a variable: <code>streamReadVar</code>	19
4.1.16. Read a variable: <code>streamReadVarF</code>	19
4.1.17. Read a horizontal slice of a variable: <code>streamReadVarSlice</code>	19
4.1.18. Read a horizontal slice of a variable: <code>streamReadVarSliceF</code>	20
4.2. Variable list functions	21
4.2.1. Create a variable list: <code>vlistCreate</code>	21
4.2.2. Destroy a variable list: <code>vlistDestroy</code>	21
4.2.3. Copy a variable list: <code>vlistCopy</code>	21
4.2.4. Duplicate a variable list: <code>vlistDuplicate</code>	21

4.2.5.	Concatenate two variable lists: <code>vlistCat</code>	22
4.2.6.	Copy some entries of a variable list: <code>vlistCopyFlag</code>	22
4.2.7.	Number of variables in a variable list: <code>vlistNvars</code>	22
4.2.8.	Number of grids in a variable list: <code>vlistNgrids</code>	22
4.2.9.	Number of zaxis in a variable list: <code>vlistNzaxis</code>	22
4.2.10.	Define the time axis: <code>vlistDefTaxis</code>	23
4.2.11.	Get the time axis: <code>vlistInqTaxis</code>	23
4.3.	Variable functions	24
4.3.1.	Define a Variable: <code>vlistDefVar</code>	24
4.3.2.	Get the Grid ID of a Variable: <code>vlistInqVarGrid</code>	25
4.3.3.	Get the Zaxis ID of a Variable: <code>vlistInqVarZaxis</code>	25
4.3.4.	Get the timestep type of a Variable: <code>vlistInqVarTsteptype</code>	25
4.3.5.	Define the code number of a Variable: <code>vlistDefVarCode</code>	26
4.3.6.	Get the Code number of a Variable: <code>vlistInqVarCode</code>	26
4.3.7.	Define the name of a Variable: <code>vlistDefVarName</code>	26
4.3.8.	Get the name of a Variable: <code>vlistInqVarName</code>	26
4.3.9.	Define the long name of a Variable: <code>vlistDefVarLongname</code>	27
4.3.10.	Get the longname of a Variable: <code>vlistInqVarLongname</code>	27
4.3.11.	Define the standard name of a Variable: <code>vlistDefVarStdname</code>	27
4.3.12.	Get the standard name of a Variable: <code>vlistInqVarStdname</code>	27
4.3.13.	Define the units of a Variable: <code>vlistDefVarUnits</code>	28
4.3.14.	Get the units of a Variable: <code>vlistInqVarUnits</code>	28
4.3.15.	Define the data type of a Variable: <code>vlistDefVarDatatype</code>	28
4.3.16.	Get the data type of a Variable: <code>vlistInqVarDatatype</code>	29
4.3.17.	Define the missing value of a Variable: <code>vlistDefVarMissval</code>	29
4.3.18.	Get the missing value of a Variable: <code>vlistInqVarMissval</code>	29
4.4.	Attributes	30
4.4.1.	Get number of attributes: <code>cdiInqNatts</code>	30
4.4.2.	Get information about an attribute: <code>cdiInqAtt</code>	30
4.4.3.	Define an integer attribute: <code>cdiDefAttInt</code>	30
4.4.4.	Get the value(s) of an integer attribute: <code>cdiInqAttInt</code>	31
4.4.5.	Define a floating point attribute: <code>cdiDefAttFlt</code>	31
4.4.6.	Get the value(s) of a floating point attribute: <code>cdiInqAttFlt</code>	31
4.4.7.	Define a text attribute: <code>cdiDefAttTxt</code>	32
4.4.8.	Get the value(s) of a text attribute: <code>cdiInqAttTxt</code>	32
4.5.	Grid functions	33
4.5.1.	Create a horizontal Grid: <code>gridCreate</code>	33
4.5.2.	Destroy a horizontal Grid: <code>gridDestroy</code>	34
4.5.3.	Duplicate a horizontal Grid: <code>gridDuplicate</code>	34
4.5.4.	Get the type of a Grid: <code>gridInqType</code>	34
4.5.5.	Get the size of a Grid: <code>gridInqSize</code>	34
4.5.6.	Define the number of values of a X-axis: <code>gridDefXsize</code>	34
4.5.7.	Get the number of values of a X-axis: <code>gridInqXsize</code>	35
4.5.8.	Define the number of values of a Y-axis: <code>gridDefYsize</code>	35
4.5.9.	Get the number of values of a Y-axis: <code>gridInqYsize</code>	35
4.5.10.	Define the number of parallels between a pole and the equator: <code>gridDefNP</code>	35
4.5.11.	Get the number of parallels between a pole and the equator: <code>gridInqNP</code>	36
4.5.12.	Define the values of a X-axis: <code>gridDefXvals</code>	36
4.5.13.	Get all values of a X-axis: <code>gridInqXvals</code>	36
4.5.14.	Define the values of a Y-axis: <code>gridDefYvals</code>	36
4.5.15.	Get all values of a Y-axis: <code>gridInqYvals</code>	37

4.5.16.	Define the bounds of a X-axis: <code>gridDefXbounds</code>	37
4.5.17.	Get the bounds of a X-axis: <code>gridInqXbounds</code>	37
4.5.18.	Define the bounds of a Y-axis: <code>gridDefYbounds</code>	37
4.5.19.	Get the bounds of a Y-axis: <code>gridInqYbounds</code>	38
4.5.20.	Define the name of a X-axis: <code>gridDefXname</code>	38
4.5.21.	Get the name of a X-axis: <code>gridInqXname</code>	38
4.5.22.	Define the longname of a X-axis: <code>gridDefXlongname</code>	38
4.5.23.	Get the longname of a X-axis: <code>gridInqXlongname</code>	39
4.5.24.	Define the units of a X-axis: <code>gridDefXunits</code>	39
4.5.25.	Get the units of a X-axis: <code>gridInqXunits</code>	39
4.5.26.	Define the name of a Y-axis: <code>gridDefYname</code>	39
4.5.27.	Get the name of a Y-axis: <code>gridInqYname</code>	40
4.5.28.	Define the longname of a Y-axis: <code>gridDefYlongname</code>	40
4.5.29.	Get the longname of a Y-axis: <code>gridInqYlongname</code>	40
4.5.30.	Define the units of a Y-axis: <code>gridDefYunits</code>	40
4.5.31.	Get the units of a Y-axis: <code>gridInqYunits</code>	41
4.5.32.	Define the reference number for an unstructured grid: <code>gridDefNumber</code>	41
4.5.33.	Get the reference number to an unstructured grid: <code>gridInqNumber</code>	41
4.5.34.	Define the position of grid in the reference file: <code>gridDefPosition</code>	41
4.5.35.	Get the position of grid in the reference file: <code>gridInqPosition</code>	41
4.5.36.	Define the reference URI for an unstructured grid: <code>gridDefReference</code>	42
4.5.37.	Get the reference URI to an unstructured grid: <code>gridInqReference</code>	42
4.5.38.	Define the UUID for an unstructured grid: <code>gridDefUUID</code>	42
4.5.39.	Get the UUID to an unstructured grid: <code>gridInqUUID</code>	42
4.6.	Z-axis functions	43
4.6.1.	Create a vertical Z-axis: <code>zaxisCreate</code>	43
4.6.2.	Destroy a vertical Z-axis: <code>zaxisDestroy</code>	44
4.6.3.	Get the type of a Z-axis: <code>zaxisInqType</code>	44
4.6.4.	Get the size of a Z-axis: <code>zaxisInqSize</code>	44
4.6.5.	Define the levels of a Z-axis: <code>zaxisDefLevels</code>	45
4.6.6.	Get all levels of a Z-axis: <code>zaxisInqLevels</code>	45
4.6.7.	Get one level of a Z-axis: <code>zaxisInqLevel</code>	45
4.6.8.	Define the name of a Z-axis: <code>zaxisDefName</code>	45
4.6.9.	Get the name of a Z-axis: <code>zaxisInqName</code>	46
4.6.10.	Define the longname of a Z-axis: <code>zaxisDefLongname</code>	46
4.6.11.	Get the longname of a Z-axis: <code>zaxisInqLongname</code>	46
4.6.12.	Define the units of a Z-axis: <code>zaxisDefUnits</code>	46
4.6.13.	Get the units of a Z-axis: <code>zaxisInqUnits</code>	47
4.7.	T-axis functions	48
4.7.1.	Create a Time axis: <code>taxisCreate</code>	48
4.7.2.	Destroy a Time axis: <code>taxisDestroy</code>	49
4.7.3.	Define the reference date: <code>taxisDefRdate</code>	49
4.7.4.	Get the reference date: <code>taxisInqRdate</code>	49
4.7.5.	Define the reference time: <code>taxisDefRtime</code>	49
4.7.6.	Get the reference time: <code>taxisInqRtime</code>	49
4.7.7.	Define the verification date: <code>taxisDefVdate</code>	50
4.7.8.	Get the verification date: <code>taxisInqVdate</code>	50
4.7.9.	Define the verification time: <code>taxisDefVtime</code>	50
4.7.10.	Get the verification time: <code>taxisInqVtime</code>	50
4.7.11.	Define the calendar: <code>taxisDefCalendar</code>	50
4.7.12.	Get the calendar: <code>taxisInqCalendar</code>	51

A. Quick Reference	53
B. Examples	68
B.1. Write a dataset	68
B.1.1. Result	69
B.2. Read a dataset	70
B.3. Copy a dataset	71
C. Environment Variables	73
Function index	74

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](http://code.zmaw.de/projects/cdi/files) 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](http://www.unidata.ucar.edu/packages/netcdf) library (<http://www.unidata.ucar.edu/packages/netcdf>) version 3 or higher. This is needed to read/write NetCDF files with **CDI**.
- ECMWF [ecCodes](https://software.ecmwf.int/wiki/display/ECC/ecCodes+Home) library (<https://software.ecmwf.int/wiki/display/ECC/ecCodes+Home>) version 2.3.0 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
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
210	–	isobaricInPa	Isobaric level in Pa

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 ecCodes [[ecCodes](#)] library. ecCodes is an external library and not part of **CDI**. To use GRIB2 with **CDI** the ecCodes library must be installed before the configuration of the **CDI** library. Use the configure option `--with-eccodes` to enable GRIB2 support.

The ecCodes library is also used to encode/decode GRIB1 records if the support for the CGRIBEX library is disabled. This feature is not tested regularly 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,ideate,itime,nlon,nlat,idispo1,idispo2
REAL*4 field(mlon,mlat)
...
READ(unit) icode,ilevel,ideate,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:

<code>icode</code>	The code number
<code>ilevel</code>	The level
<code>ideate</code>	The date as YYYYMMDD
<code>itime</code>	The time as hhmmss
<code>nlon</code>	The number of longitudes
<code>nlat</code>	The number of latitudes
<code>idispo1</code>	For the users disposal (Not used in CDI)
<code>idispo2</code>	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 ideate,icode,ilevel,nsize
REAL*4 field(msize)
...
READ(unit) ideate,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 variable
    ...
    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:

<code>CDI_FILETYPE_GRB</code>	File type GRIB version 1
<code>CDI_FILETYPE_GRB2</code>	File type GRIB version 2
<code>CDI_FILETYPE_NC</code>	File type NetCDF
<code>CDI_FILETYPE_NC2</code>	File type NetCDF version 2 (64-bit offset)
<code>CDI_FILETYPE_NC4</code>	File type NetCDF-4 (HDF5)
<code>CDI_FILETYPE_NC4C</code>	File type NetCDF-4 classic
<code>CDI_FILETYPE_NC5</code>	File type NetCDF version 5 (64-bit data)
<code>CDI_FILETYPE_SRV</code>	File type SERVICE
<code>CDI_FILETYPE_EXT</code>	File type EXTRA
<code>CDI_FILETYPE_IEG</code>	File type IEG

`CDI_FILETYPE_GRB2` is only available if the **CDI** library was compiled with ecCodes 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 `CDI_FILETYPE_SRV`, `CDI_FILETYPE_EXT` or `CDI_FILETYPE_IEG` use one of the following predefined constants in the call to `streamDefByteorderstreamDef`:

<code>CDI_BIGENDIAN</code>	Byte order big endian
<code>CDI_LITTLEENDIAN</code>	Byte order little endian

4.1.1. Create a new dataset: `streamOpenWrite`

The function `streamOpenWrite` creates a new dataset.

Usage

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

<code>path</code>	The name of the new dataset.
<code>filetype</code>	The type of the file format, one of the set of predefined CDI file format types. The valid CDI file format types are <code>CDI_FILETYPE_GRB</code> , <code>CDI_FILETYPE_GRB2</code> , <code>CDI_FILETYPE_NC</code> , <code>CDI_FILETYPE_NC2</code> , <code>CDI_FILETYPE_NC4</code> , <code>CDI_FILETYPE_NC4C</code> , <code>CDI_FILETYPE_NC5</code> , <code>CDI_FILETYPE_SRV</code> , <code>CDI_FILETYPE_EXT</code> and <code>CDI_FILETYPE_IEG</code> .

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", CDI_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 `CDI_FILETYPE_GRB`, `CDI_FILETYPE_GRB2`, `CDI_FILETYPE_NC`, `CDI_FILETYPE_NC2`, `CDI_FILETYPE_NC4`, `CDI_FILETYPE_NC4C`, `CDI_FILETYPE_NC5`, `CDI_FILETYPE_SRV`, `CDI_FILETYPE_EXT` and `CDI_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 `CDI_FILETYPE_SRV`, `CDI_FILETYPE_EXT` or `CDI_FILETYPE_IEG`.

Usage

```
void streamDefByteorder(int streamID, int 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 `CDI_FILETYPE_SRV`, `CDI_FILETYPE_EXT` or `CDI_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.

To safeguard against errors by modifying the wrong vlist object, this function makes the passed vlist object immutable. All further vlist changes have to use the vlist object returned by `streamInqVlist()`.

Usage

```
void streamDefVlist(int streamID, int 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

```
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 `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. The values are converted to the external data type of the variable, if necessary.

Usage

```
void streamWriteVar(int streamID, int varID, const double *data, int 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.

Usage

```
void streamWriteVarF(int streamID, int varID, const float *data, int 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. 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

```
void streamWriteVarSlice(int streamID, int varID, int levelID, const double *data,  
                        int 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.

4.1.14. 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.

Usage

```
void streamWriteVarSliceF(int streamID, int varID, int levelID, const float *data,
                          int 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.

4.1.15. 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](#).
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.

4.1.16. Read a variable: streamReadVarF

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, float *data, int *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.

4.1.17. 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

```
void streamReadVarSlice(int streamID, int varID, int levelID, double *data,
                        int *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.

4.1.18. Read a horizontal slice of a variable: `streamReadVarSliceF`

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

Usage

```
void streamReadVarSliceF(int streamID, int varID, int levelID, float *data,  
                        int *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

```
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, TSTEP_INSTANT);
...
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` 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

```
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](#) 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

```
int vlistNgrids(int 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

```
int vlistNzaxis(int 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

```
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](#) 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 timestep types must be specified:

<code>TSTEP_CONSTANT</code>	The data values have no time dimension.
<code>TSTEP_INSTANT</code>	The data values are representative of points in space or time (instantaneous).
<code>TSTEP_ACCUM</code>	The data values are representative of a sum or accumulation over the cell.
<code>TSTEP_AVG</code>	Mean (average value)
<code>TSTEP_MAX</code>	Maximum
<code>TSTEP_MIN</code>	Minimum
<code>TSTEP_SD</code>	Standard deviation

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:

<code>CDI_DATATYPE_PACK8</code>	8 packed bit (only for GRIB)
<code>CDI_DATATYPE_PACK16</code>	16 packed bit (only for GRIB)
<code>CDI_DATATYPE_PACK24</code>	24 packed bit (only for GRIB)
<code>CDI_DATATYPE_FLT32</code>	32 bit floating point
<code>CDI_DATATYPE_FLT64</code>	64 bit floating point
<code>CDI_DATATYPE_INT8</code>	8 bit integer
<code>CDI_DATATYPE_INT16</code>	16 bit integer
<code>CDI_DATATYPE_INT32</code>	32 bit integer

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 tsteptype);
```

<code>vlistID</code>	Variable list ID, from a previous call to <code>vlistCreate</code> .
<code>gridID</code>	Grid ID, from a previous call to <code>gridCreate</code> .
<code>zaxisID</code>	Z-axis ID, from a previous call to <code>zaxisCreate</code> .
<code>tsteptype</code>	One of the set of predefined CDI timestep types. The valid CDI timestep types are <code>TSTEP_CONSTANT</code> , <code>TSTEP_INSTANT</code> , <code>TSTEP_ACCUM</code> , <code>TSTEP_AVG</code> , <code>TSTEP_MAX</code> , <code>TSTEP_MIN</code> and <code>TSTEP_SD</code> .

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_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

```
int vlistInqVarGrid(int vlistID, int 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

```
int vlistInqVarZaxis(int vlistID, int 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. Get the timestep type of a Variable: `vlistInqVarTsteptype`

The function `vlistInqVarTsteptype` returns the timestep type of a Variable.

Usage

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

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

Result

`vlistInqVarTstepType` returns the timestep type of the Variable, one of the set of predefined **CDI** timestep types. The valid **CDI** timestep types are `TSTEP_CONSTANT`, `TSTEP_INSTANT`, `TSTEP_ACCUM`, `TSTEP_AVG`, `TSTEP_MAX`, `TSTEP_MIN` and `TSTEP_SD`.

4.3.5. 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.
```

4.3.6. 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 or streamInqVlist.  
varID    Variable identifier.
```

Result

`vlistInqVarCode` returns the code number of the variable.

4.3.7. 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.
```

4.3.8. 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 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 pre-  
          defined 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.

4.3.9. 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.

4.3.10. 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

```
void vlistInqVarLongname(int vlistID, int varID, char *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`.

4.3.11. 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.

4.3.12. 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

```
void vlistInqVarStdname(int vlistID, int varID, char *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

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

4.3.13. 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.

4.3.14. 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

```
void vlistInqVarUnits(int vlistID, int varID, char *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`.

4.3.15. 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 `CDI_DATATYPE_PACK8`, `CDI_DATATYPE_PACK16`, `CDI_DATATYPE_PACK24`, `CDI_DATATYPE_FLT32`, `CDI_DATATYPE_FLT64`, `CDI_DATATYPE_INT8`, `CDI_DATATYPE_INT16` and `CDI_DATATYPE_INT32`.

4.3.16. 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](#) or [streamInqVlist](#).

varID Variable identifier.

Result

`vlistInqVarDatatype` returns an identifier to the data type of the variable. The valid **CDI** data types are `CDI_DATATYPE_PACK8`, `CDI_DATATYPE_PACK16`, `CDI_DATATYPE_PACK24`, `CDI_DATATYPE_FLT32`, `CDI_DATATYPE_FLT64`, `CDI_DATATYPE_INT8`, `CDI_DATATYPE_INT16` and `CDI_DATATYPE_INT32`.

4.3.17. 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.

4.3.18. 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](#) 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:

<code>CDI_DATATYPE_INT16</code>	16-bit integer attribute
<code>CDI_DATATYPE_INT32</code>	32-bit integer attribute
<code>CDI_DATATYPE_FLT32</code>	32-bit floating point attribute
<code>CDI_DATATYPE_FLT64</code>	64-bit floating point attribute
<code>CDI_DATATYPE_TXT</code>	Text attribute

4.4.1. Get number of attributes: `cdiInqNatts`

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

Usage

```
int cdiInqNatts(int cdiID, int varID, int *nattsp);
```

`cdiID` CDI ID, from a previous call to [vlistCreate](#), [gridCreate](#) or [streamInqVlist](#).

`varID` Variable identifier, or `CDI_GLOBAL` for a global attribute.

`nattsp` Pointer to location for returned number of attributes.

4.4.2. Get information about an attribute: `cdiInqAtt`

The function `cdiInqAtt` gets information about an attribute.

Usage

```
int cdiInqAtt(int cdiID, int varID, int attnum, char *name, int *typep, int *lenp);
```

`cdiID` CDI ID, from a previous call to [vlistCreate](#), [gridCreate](#) 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: `cdiDefAttInt`

The function `cdiDefAttInt` defines an integer attribute.

Usage

```
int cdiDefAttInt(int cdiID, int varID, const char *name, int type, int len,
                 const int *ip);
```

cdiID CDI ID, from a previous call to [vlistCreate](#) or [gridCreate](#).
varID Variable identifier, or `CDI_GLOBAL` for a global attribute.
name Attribute name.
type External data type (`CDI_DATATYPE_INT16` or `CDI_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: cdiInqAttInt

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

Usage

```
int cdiInqAttInt(int cdiID, int varID, const char *name, int mlen, int *ip);
```

cdiID CDI ID, from a previous call to [vlistCreate](#), [gridCreate](#) 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: cdiDefAttFlt

The function `cdiDefAttFlt` defines a floating point attribute.

Usage

```
int cdiDefAttFlt(int cdiID, int varID, const char *name, int type, int len,
                 const double *dp);
```

cdiID CDI ID, from a previous call to [vlistCreate](#) or [gridCreate](#).
varID Variable identifier, or `CDI_GLOBAL` for a global attribute.
name Attribute name.
type External data type (`CDI_DATATYPE_FLT32` or `CDI_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: cdiInqAttFlt

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

Usage

```
int cdiInqAttFlt(int cdiID, int varID, const char *name, int mlen, double *dp);
```

cdiID CDI ID, from a previous call to [vlistCreate](#), [gridCreate](#) 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: `cdiDefAttTxt`

The function `cdiDefAttTxt` defines a text attribute.

Usage

```
int cdiDefAttTxt(int cdiID, int varID, const char *name, int len, const char *tp);
```

cdiID CDI ID, from a previous call to [vlistCreate](#) or [gridCreate](#).
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: `cdiInqAttTxt`

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

Usage

```
int cdiInqAttTxt(int cdiID, int varID, const char *name, int mlen, char *tp);
```

cdiID CDI ID, from a previous call to [vlistCreate](#), [gridCreate](#) 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_LCC	Lambert conformal conic grid
GRID_CURVILINEAR	Curvilinear grid
GRID_UNSTRUCTURED	Unstructured grid

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_PROJECTION`, `GRID_SPECTRAL`, `GRID_GME`, `GRID_CURVILINEAR` and `GRID_UNSTRUCTURED`.

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](#) 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](#) 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_PROJECTION`, `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

```
int gridInqSize(int 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

```
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

```
int gridInqXsize(int 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

```
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

```
int gridInqYsize(int 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

```
void gridDefNP(int gridID, int 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

```
int gridInqNP(int 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

```
void gridDefXvals(int gridID, const double *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

```
int gridInqXvals(int gridID, double *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

```
void gridDefYvals(int gridID, const double *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

```
int gridInqYvals(int gridID, double *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

```
void gridDefXbounds(int gridID, const double *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

```
int gridInqXbounds(int gridID, double *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

```
void gridDefYbounds(int gridID, const double *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

```
int gridInqYbounds(int gridID, double *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

```
void gridDefXname(int gridID, const char *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

```
void gridInqXname(int gridID, char *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

```
void gridDefXlongname(int gridID, const char *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

```
void gridInqXlongname(int gridID, char *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

```
void gridDefXunits(int gridID, const char *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

```
void gridInqXunits(int gridID, char *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

```
void gridDefYname(int gridID, const char *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

```
void gridInqYname(int gridID, char *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

```
void gridDefYlongname(int gridID, const char *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

```
void gridInqYlongname(int gridID, char *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

```
void gridDefYunits(int gridID, const char *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

```
void gridInqYunits(int gridID, char *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

```
void gridDefNumber(int gridID, const int 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

```
int gridInqNumber(int 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

```
void gridDefPosition(int gridID, const int 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

```
int gridInqPosition(int 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

```
void gridDefReference(int gridID, const char *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(int gridID, char *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

```
void gridDefUUID(int gridID, const char *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

```
void gridInqUUID(int gridID, char *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

```
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_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
...
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](#) 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_BOTTOM_TB`, `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

```
int zaxisInqSize(int 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

```
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` 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

```
double zaxisInqLevel(int zaxisID, int 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

```
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` 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

```
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` 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

```
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` 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 assigned to a variable list using `vlistDefTaxis`. The following different Time axis types are available:

<code>TAXIS_ABSOLUTE</code>	Absolute time axis
<code>TAXIS_RELATIVE</code>	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:

<code>CALENDAR_STANDARD</code>	Mixed Gregorian/Julian calendar.
<code>CALENDAR_PROLEPTIC</code>	Proleptic Gregorian calendar. This is the default.
<code>CALENDAR_360DAYS</code>	All years are 360 days divided into 30 day months.
<code>CALENDAR_365DAYS</code>	Gregorian calendar without leap years, i.e., all years are 365 days long.
<code>CALENDAR_366DAYS</code>	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 `taxisDefRdate` 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` 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

```
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` 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

```
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](#) 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

```
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](#) 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

```
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](#) 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

[ecCodes]

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

[ECHAM]

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

[GRIB]

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

[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 C language bindings of the **CDI** library routines:

`cdiDefAttFlt`

```
int cdiDefAttFlt(int cdiID, int varID, const char *name, int type, int len,
                 const double *dp);
```

Define a floating point attribute

`cdiDefAttInt`

```
int cdiDefAttInt(int cdiID, int varID, const char *name, int type, int len,
                 const int *ip);
```

Define an integer attribute

`cdiDefAttTxt`

```
int cdiDefAttTxt(int cdiID, int varID, const char *name, int len, const char *tp);
```

Define a text attribute

`cdiInqAtt`

```
int cdiInqAtt(int cdiID, int varID, int attnum, char *name, int *typep, int *lenp);
```

Get information about an attribute

`cdiInqAttFlt`

```
int cdiInqAttFlt(int cdiID, int varID, const char *name, int mlen, double *dp);
```

Get the value(s) of a floating point attribute

`cdiInqAttInt`

```
int cdiInqAttInt(int cdiID, int varID, const char *name, int mlen, int *ip);
```

Get the value(s) of an integer attribute

`cdiInqAttTxt`

```
int cdiInqAttTxt(int cdiID, int varID, const char *name, int mlen, char *tp);
```

Get the value(s) of a text attribute

`cdiInqNatts`

```
int cdiInqNatts(int cdiID, int varID, int *nattsp);
```

Get number of attributes

`gridCreate`

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

Create a horizontal Grid

`gridDefNP`

```
void gridDefNP(int gridID, int np);
```

Define the number of parallels between a pole and the equator

`gridDefNumber`

```
void gridDefNumber(int gridID, const int number);
```

Define the reference number for an unstructured grid

`gridDefPosition`

```
void gridDefPosition(int gridID, const int position);
```

Define the position of grid in the reference file

`gridDefReference`

```
void gridDefReference(int gridID, const char *reference);
```

Define the reference URI for an unstructured grid

`gridDefUUID`

```
void gridDefUUID(int gridID, const char *uuid);
```

Define the UUID for an unstructured 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

`gridInqNP`

```
int gridInqNP(int gridID);
```

Get the number of parallels between a pole and the equator

`gridInqNumber`

```
int gridInqNumber(int gridID);
```

Get the reference number to an unstructured grid

`gridInqPosition`

```
int gridInqPosition(int gridID);
```

Get the position of grid in the reference file

gridInqReference

```
char *gridInqReference(int gridID, char *reference);
```

Get the reference URI to an unstructured grid

gridInqSize

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

Get the size of a Grid

gridInqType

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

Get the type of a Grid

gridInqUUID

```
void gridInqUUID(int gridID, char *uuid);
```

Get the UUID to an unstructured grid

gridInqXbounds

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

Get the bounds of a X-axis

gridInqXlongname

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

Get the longname of a X-axis

gridInqXname

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

Get the name of a X-axis

gridInqXsize

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

Get the number of values of a X-axis

`gridInqXunits`

```
void gridInqXunits(int gridID, 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 gridInqYlongname(int gridID, char *longname);
```

Get the longname of a Y-axis

`gridInqYname`

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

Get the name of a Y-axis

`gridInqYsize`

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

Get the number of values of a Y-axis

`gridInqYunits`

```
void gridInqYunits(int gridID, 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

`streamDefRecord`

```
void streamDefRecord(int streamID, int varID, int levelID);
```

Define the next record

`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

`streamReadVarF`

```
void streamReadVar(int streamID, int varID, float *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

`streamReadVarSliceF`

```
void streamReadVarSliceF(int streamID, int varID, int levelID, float *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

`streamWriteVarF`

```
void streamWriteVarF(int streamID, int varID, const float *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

`streamWriteVarSliceF`

```
void streamWriteVarSliceF(int streamID, int varID, int levelID, const float *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

`vlistDefTaxis`

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

Define the time axis

`vlistDefVar`

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

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

`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

`vlistInqVarTsteptype`

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

Get the timestep type 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.

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

int main(void)
5 {
    const int nlon = 12; // Number of longitudes
    const int nlat = 6;  // Number of latitudes
    const int nlev = 5;  // Number of levels
    const int nts = 3;   // Number of time steps
10    int nmiss = 0;
    double lons[] = {0, 30, 60, 90, 120, 150, 180, 210, 240, 270, 300, 330};
    double lats[] = {-75, -45, -15, 15, 45, 75};
    double levs[] = {101300, 92500, 85000, 50000, 20000};
15    double var1[nlon*nlat];
    double var2[nlon*nlat*nlev];

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

    // Create a surface level Z-axis
25    int zaxisID1 = zaxisCreate(ZAXIS_SURFACE, 1);

    // Create a pressure level Z-axis
    int zaxisID2 = zaxisCreate(ZAXIS_PRESSURE, nlev);
30    zaxisDefLevels(zaxisID2, levs);

    // Create a variable list
    int vlistID = vlistCreate();

    // Define the variables
35    int varID1 = vlistDefVar(vlistID, gridID, zaxisID1, TSTEP_INSTANT);
    int varID2 = vlistDefVar(vlistID, gridID, zaxisID2, TSTEP_INSTANT);

    // Define the variable names
40    vlistDefVarName(vlistID, varID1, "varname1");
    vlistDefVarName(vlistID, varID2, "varname2");
```

```

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

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

    // Create a dataset in netCDF format
50 int streamID = streamOpenWrite("example.nc", CDI_FILETYPE_NC);
    if ( streamID < 0 )
    {
        fprintf(stderr, "%s\n", cdiStringError(streamID));
        return 1;
55     }

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

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

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

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

    // Close the output stream
80 streamClose(streamID);

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

    return 0;
90 }

```

B.1.1. Result

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

```

netcdf example {
dimensions:
    lon = 12 ;

```

```

5      lat = 6 ;
      lev = 5 ;
      time = UNLIMITED ; // (3 currently)
variables :
      double lon(lon) ;
          lon:long_name = "longitude" ;
10         lon:units = "degrees_east" ;
          lon:standard_name = "longitude" ;
      double lat(lat) ;
          lat:long_name = "latitude" ;
          lat:units = "degrees_north" ;
15         lat:standard_name = "latitude" ;
      double lev(lev) ;
          lev:long_name = "pressure" ;
          lev:units = "Pa" ;
      double time(time) ;
20         time:units = "day as %Y%m%d.%f" ;
      float varname1(time, lat, lon) ;
      float varname2(time, lev, lat, lon) ;
data:
25 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 ;
30 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"
3
int main(void)
{
    const int nlon = 12; // Number of longitudes
    const int nlat = 6;  // Number of latitudes
8    const int nlev = 5; // Number of levels
    const int nts = 3;  // Number of time steps
    int nmiss, vdate, vtime;
    double var1[nlon*nlat];
    double var2[nlon*nlat*nlev];
13

    // Open the dataset
    int streamID = streamOpenRead("example.nc");
    if ( streamID < 0 )
18 {
        fprintf(stderr, "%s\n", cdiStringError(streamID));
        return 1;
    }
}

```

```

23 // Get the variable list of the dataset
    int vlistID = streamInqVlist(streamID);

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

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

33 // Loop over the number of time steps
    for ( int tsID = 0; tsID < nts; tsID++ )
    {
        // Inquire the time step
        streamInqTimestep(streamID, tsID);
38

        // Get the verification date and time
        vdate = taxisInqVdate(taxisID);
        vtime = taxisInqVtime(taxisID);
        printf("read timestep %d: date=%d time=%d\n", tsID+1, vdate, vtime);
43

        // Read var1 and var2
        streamReadVar(streamID, varID1, var1, &nmiss);
        streamReadVar(streamID, varID2, var2, &nmiss);
    }
48

    // Close the input stream
    streamClose(streamID);

    return 0;
53 }

```

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 `CDI_FILETYPE_GRB`.

```

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

int main(void)
{
    const int nlon = 12; // Number of longitudes
7    const int nlat = 6; // Number of latitudes
    const int nlev = 5; // Number of levels
    const int nts = 3; // Number of time steps
    int nmiss;
    double var1[nlon*nlat];
12    double var2[nlon*nlat*nlev];

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

```

```

    return 1;
}
22
// Get the variable list of the dataset
int vlistID1 = streamInqVlist(streamID1);

// Set the variable IDs
27
int varID1 = 0;
int varID2 = 1;

// Open the output dataset (GRIB format)
int streamID2 = streamOpenWrite("example.grb", CDI_FILETYPE_GRB);
32
if ( streamID2 < 0 )
{
    fprintf(stderr, "%s\n", cdiStringError(streamID2));
    return 1;
}
37
int vlistID2 = vlistDuplicate(vlistID1);

streamDefVlist(streamID2, vlistID2);

// Loop over the number of time steps
42
for ( int tsID = 0; tsID < nts; tsID++ )
{
    // Inquire the input time step
    streamInqTimestep(streamID1, tsID);
47

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

    // Read var1 and var2
52
    streamReadVar(streamID1, varID1, var1, &nmiss);
    streamReadVar(streamID1, varID2, var2, &nmiss);

    // Write var1 and var2
    streamWriteVar(streamID2, varID1, var1, nmiss);
57
    streamWriteVar(streamID2, varID2, var2, nmiss);
}

// Close the streams
streamClose(streamID1);
62
streamClose(streamID2);

return 0;
}

```


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

C

cdiDefAttFlt	31
cdiDefAttInt	30
cdiDefAttTxt	32
cdiInqAtt	30
cdiInqAttFlt	31
cdiInqAttInt	31
cdiInqAttTxt	32
cdiInqNatts	30

G

gridCreate	33
gridDefNP	35
gridDefNumber	41
gridDefPosition	41
gridDefReference	42
gridDefUUID	42
gridDefXbounds	37
gridDefXlongname	38
gridDefXname	38
gridDefXsize	34
gridDefXunits	39
gridDefXvals	36
gridDefYbounds	37
gridDefYlongname	40
gridDefYname	39
gridDefYsize	35
gridDefYunits	40
gridDefYvals	36
gridDestroy	34
gridDuplicate	34
gridInqNP	36
gridInqNumber	41
gridInqPosition	41
gridInqReference	42
gridInqSize	34
gridInqType	34
gridInqUUID	42
gridInqXbounds	37
gridInqXlongname	39
gridInqXname	38
gridInqXsize	35
gridInqXunits	39

gridInqXvals	36
gridInqYbounds	38
gridInqYlongname	40
gridInqYname	40
gridInqYsize	35
gridInqYunits	41
gridInqYvals	37

S

streamClose	16
streamDefByteorder	16
streamDefTimestep	17
streamDefVlist	17
streamInqByteorder	16
streamInqFiletype	16
streamInqTimestep	17
streamInqVlist	17
streamOpenRead	15
streamOpenWrite	14
streamReadVar	19
streamReadVarF	19
streamReadVarSlice	19
streamReadVarSliceF	20
streamWriteVar	18
streamWriteVarF	18
streamWriteVarSlice	18
streamWriteVarSliceF	18

T

taxisCreate	48
taxisDefCalendar	50
taxisDefRdate	49
taxisDefRtime	49
taxisDefVdate	50
taxisDefVtime	50
taxisDestroy	49
taxisInqCalendar	51
taxisInqRdate	49
taxisInqRtime	49
taxisInqVdate	50
taxisInqVtime	50

V

vlistCat	22
--------------------------	--------------------

vlistCopy	21
vlistCopyFlag	22
vlistCreate	21
vlistDefTaxis	23
vlistDefVar	24
vlistDefVarCode	26
vlistDefVarDatatype	28
vlistDefVarLongname	27
vlistDefVarMissval	29
vlistDefVarName	26
vlistDefVarStdname	27
vlistDefVarUnits	28
vlistDestroy	21
vlistDuplicate	21
vlistInqTaxis	23
vlistInqVarCode	26
vlistInqVarDatatype	29
vlistInqVarGrid	25
vlistInqVarLongname	27
vlistInqVarMissval	29
vlistInqVarName	26
vlistInqVarStdname	27
vlistInqVarTsteptype	25
vlistInqVarUnits	28
vlistInqVarZaxis	25
vlistNgrids	22
vlistNvars	22
vlistNzaxis	22

Z

zaxisCreate	43
zaxisDefLevels	45
zaxisDefLongname	46
zaxisDefName	45
zaxisDefUnits	46
zaxisDestroy	44
zaxisInqLevel	45
zaxisInqLevels	45
zaxisInqLongname	46
zaxisInqName	46
zaxisInqSize	44
zaxisInqType	44
zaxisInqUnits	47