# CDI Fortran Manual

**Uwe Schulzweida**
**Max-Planck-Institute for Meteorology**

# Contents

# 1. Introduction

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

## 1.1. Building from sources

This section describes how to build the **CDI** library from the sources on a UNIX system. **CDI** is using the GNU configure and build system to compile the source code. The only requirement is a working ANSI C99 compiler.

First go to the download page (`https://code.mpimet.mpg.de/projects/cdi/files`) to get the latest distribution, if you do not already have it.

To take full advantage of **CDI**'s features the following additional libraries should be installed:

- Unidata NetCDF library (`http://www.unidata.ucar.edu/packages/netcdf`) version 3 or higher. This is needed to read/write NetCDF files with **CDI**.

- ECMWF ecCodes 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.

The GRIB records must be sorted by time to be able to read them correctly with **CDI**.

**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 a 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 regulary and the status is experimental!

A single GRIB2 message can contain multiple fields. This feature is not supported in **CDI**!

## 2.2. NetCDF

NetCDF [NetCDF] (Network Common Data Form) is an interface for array-oriented data access and a library that provides an implementation of the interface. The NetCDF library also defines

a machine-independent format for representing scientific data. Together, the interface, library, and format support the creation, access, and sharing of scientific data.

**CDI** only supports the classic data model of NetCDF and arrays up to 4 dimensions. These dimensions should only be used by the horizontal and vertical grid and the time. The NetCDF attributes should follow the GDT, COARDS or CF Conventions.

NetCDF is an external library and not part of **CDI**. To use NetCDF with **CDI** the NetCDF library must be installed before the configuration of the **CDI** library. Use the configure option `--with-netcdf` to enable NetCDF support (see Build).

## 2.3. SERVICE

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

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

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

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

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

## 2.4. EXTRA

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

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

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

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

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

## 2.5.  IEG

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

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

# 3. Use of the CDI Library

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

## 3.1. Creating a dataset

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

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

## 3.2. Reading a dataset

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

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

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

## 3.3. Compiling and Linking with the CDI library

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

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

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

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

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

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

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

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

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

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

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

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

```
    USE ISO_C_BINDING
    USE mo_cdi
```

The `iso_c_binding` module is included in `mo_cdi`, but without `cfortran.h` characters and character variables have to be handled separately. Examples are available in section B.4.

After installation `mo_cdi.o` and `mo_cdi.mod` are located in the library and header directory respectively. `cdilib.o` has to be mentioned directly on the command line. It can be found in the library directory, too. Depending on the **CDI** configuration, a compile command should look like this:

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

# 4. CDI modules

## 4.1. Dataset functions

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

| | |
|---|---|
| CDI_FILETYPE_GRB | File type GRIB version 1 |
| CDI_FILETYPE_GRB2 | File type GRIB version 2 |
| CDI_FILETYPE_NC | File type NetCDF |
| CDI_FILETYPE_NC2 | File type NetCDF version 2 (64-bit offset) |
| CDI_FILETYPE_NC4 | File type NetCDF-4 (HDF5) |
| CDI_FILETYPE_NC4C | File type NetCDF-4 classic |
| CDI_FILETYPE_NC5 | File type NetCDF version 5 (64-bit data) |
| CDI_FILETYPE_NCZARR | File type NetCDF NCZarr |
| CDI_FILETYPE_SRV | File type SERVICE |
| CDI_FILETYPE_EXT | File type EXTRA |
| CDI_FILETYPE_IEG | 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 streamDefByteorder:

| | |
|---|---|
| CDI_BIGENDIAN | Byte order big endian |
| CDI_LITTLEENDIAN | Byte order little endian |

### 4.1.1. Create a new dataset: streamOpenWrite

The function streamOpenWrite creates a new datset.

**Usage**

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

path   The name of the new dataset.

filetype  The type of the file format, one of the set of predefined **CDI** file format types. The valid **CDI** file format types are CDI_FILETYPE_GRB, CDI_FILETYPE_GRB2, CDI_FILETYPE_NC, CDI_FILETYPE_NC2, CDI_FILETYPE_NC4, CDI_FILETYPE_NC4C, CDI_FILETYPE_NC5, CDI_FILETYPE_NCZARR, CDI_FILETYPE_SRV, CDI_FILETYPE_EXT and CDI_FILETYPE_IEG.

**Result**

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

**Errors**

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

**Example**

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

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

### 4.1.2. Open a dataset for reading: `streamOpenRead`

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

**Usage**

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

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

**Result**

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

**Errors**

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

**Example**

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

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

### 4.1.3. Close an open dataset: `streamClose`

The function `streamClose` closes an open dataset.

**Usage**

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

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

### 4.1.4. Get the filetype: `streamInqFiletype`

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

**Usage**

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

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

**Result**

`streamInqFiletype` returns the type of the file format, one of the set of predefined **CDI** file format types. The valid **CDI** file format types are CDI_FILETYPE_GRB, CDI_FILETYPE_GRB2, CDI_FILETYPE_NC, CDI_FILETYPE_NC2, CDI_FILETYPE_NC4, CDI_FILETYPE_NC4C, CDI_FILETYPE_NC5, CDI_FILETYPE_NCZARR, 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**

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

    `streamID`    Stream ID, from a previous call to streamOpenWrite.
    `byteorder`   The byte order of a dataset, one of the **CDI** constants CDI_BIGENDIAN and CDI_LITTLEENDIAN.

### 4.1.6. Get the byte order: `streamInqByteorder`

The function `streamInqByteorder` returns the byte order of a binary dataset with the file format type CDI_FILETYPE_SRV, CDI_FILETYPE_EXT or CDI_FILETYPE_IEG.

**Usage**

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

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

**Result**

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

### 4.1.7. Define the variable list: `streamDefVlist`

The function `streamDefVlist` defines the variable list of a stream.
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 stream-InqVlist().

**Usage**

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

  streamID   Stream ID, from a previous call to streamOpenWrite.

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

### 4.1.8. Get the variable list: `streamInqVlist`

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

**Usage**

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

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

**Result**

`streamInqVlist` returns an identifier to the variable list.

### 4.1.9. Define a timestep: `streamDefTimestep`

The function `streamDefTimestep` defines a timestep of a stream by the identifier tsID. The identifier tsID is the timestep index starting at 0 for the first timestep. Before calling this function the functions `taxisDefVdate` and `taxisDefVtime` should be used to define the timestamp for this timestep. All calls to write the data refer to this timestep.

**Usage**

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

  streamID   Stream ID, from a previous call to streamOpenWrite.

  tsID      Timestep identifier.

**Result**

`streamDefTimestep` returns the number of expected records of the timestep.

### 4.1.10. Get timestep information: `streamInqTimestep`

The function `streamInqTimestep` sets the next timestep to the identifier tsID. The identifier tsID is the timestep index starting at 0 for the first timestep. After a call to this function the functions `taxisInqVdate` and `taxisInqVtime` can be used to read the timestamp for this timestep. All calls to read the data refer to this timestep.

**Usage**

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

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

tsID        Timestep identifier.

**Result**

`streamInqTimestep` returns the number of records of the timestep or 0, if the end of the file is reached.

### 4.1.11. Write a variable: `streamWriteVar`

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

**Usage**

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

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.

numMissVals   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**

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

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.

numMissVals   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**

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

| | |
|---|---|
| `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. |
| `numMissVals` | 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**

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

| | |
|---|---|
| `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. |
| `numMissVals` | 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**

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

| | |
|---|---|
| `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. |
| `numMissVals` | 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**

```
    SUBROUTINE streamReadVar(INTEGER streamID, INTEGER varID, REAL*4 data,
                             INTEGER numMissVals)
```

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.

numMissVals   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**

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

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.

numMissVals   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**

```
    SUBROUTINE streamReadVarSliceF(INTEGER streamID, INTEGER varID, INTEGER levelID,
                                   REAL*4 data, INTEGER numMissVals)
```

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.

numMissVals   Number of missing values.

## 4.2. Variable list functions

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

### 4.2.1. Create a variable list: `vlistCreate`

**Usage**

```
INTEGER FUNCTION vlistCreate()
```

**Example**

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

> **INCLUDE** 'cdi.inc'
>     ...
> **INTEGER** vlistID, varID
>     ...
> vlistID  = vlistCreate()
> varID = vlistDefVar(vlistID, gridID, zaxisID, TIME_VARYING)
>     ...
> streamDefVlist(streamID, vlistID)
>     ...
> vlistDestroy (vlistID)
>     ...

### 4.2.2. Destroy a variable list: `vlistDestroy`

**Usage**

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

`vlistID`  Variable list ID, from a previous call to [vlistCreate](vlistCreate).

### 4.2.3. Copy a variable list: `vlistCopy`

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

**Usage**

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

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

### 4.2.4. Duplicate a variable list: `vlistDuplicate`

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

**Usage**

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

`vlistID`  Variable list ID, from a previous call to [vlistCreate](vlistCreate) or [streamInqVlist](streamInqVlist).

**Result**

vlistDuplicate returns an identifier to the duplicated variable list.

### 4.2.5. Concatenate two variable lists: vlistCat

Concatenate the variable list vlistID1 at the end of vlistID2.

**Usage**

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

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

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

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

**Usage**

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

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

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

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

**Usage**

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

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

**Result**

vlistNvars returns the number of variables in a variable list.

### 4.2.8. Number of grids in a variable list: vlistNumGrids

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

**Usage**

```
INTEGER FUNCTION vlistNumGrids(INTEGER vlistID)
```

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

**Result**

vlistNumGrids returns the number of grids in a variable list.

### 4.2.9. Number of zaxis in a variable list: vlistNumZaxis

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

**Usage**

```
    INTEGER FUNCTION vlistNumZaxis(INTEGER vlistID)
```

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

**Result**

vlistNumZaxis returns the number of zaxis in a variable list.

### 4.2.10. Define the time axis: vlistDefTaxis

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

**Usage**

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

vlistID   Variable list ID, from a previous call to vlistCreate.
taxisID   Time axis ID, from a previous call to taxisCreate.

### 4.2.11. Get the time axis: vlistInqTaxis

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

**Usage**

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

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

**Result**

vlistInqTaxis returns an identifier to the time axis.

## 4.3.  Variable functions

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

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

| | |
|---|---|
| CDI_DATATYPE_PACK8 | 8 packed bit (only for GRIB) |
| CDI_DATATYPE_PACK16 | 16 packed bit (only for GRIB) |
| CDI_DATATYPE_PACK24 | 24 packed bit (only for GRIB) |
| CDI_DATATYPE_FLT32 | 32 bit floating point |
| CDI_DATATYPE_FLT64 | 64 bit floating point |
| CDI_DATATYPE_INT8 | 8 bit integer |
| CDI_DATATYPE_INT16 | 16 bit integer |
| CDI_DATATYPE_INT32 | 32 bit integer |

### 4.3.1.  Define a Variable: `vlistDefVar`

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

**Usage**

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

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

**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.inc'
        ...
    INTEGER vlistID, varID
        ...
    vlistID = vlistCreate()
    varID = vlistDefVar(vlistID, gridID, zaxisID, TIME_VARYING)
        ...
    streamDefVlist(streamID, vlistID)
        ...
    vlistDestroy(vlistID)
        ...
```

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

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

**Usage**

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

vlistID   Variable list ID, from a previous call to [vlistCreate](#) or [streamInqVlist](#).
varID     Variable identifier.

**Result**

`vlistInqVarGrid` returns the grid ID of the Variable.

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

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

**Usage**

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

vlistID   Variable list ID, from a previous call to [vlistCreate](#) or [streamInqVlist](#).
varID     Variable identifier.

**Result**

`vlistInqVarZaxis` returns the zaxis ID of the variable.

### 4.3.4. Get the timestep type of a Variable: `vlistInqVarTsteptype`

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

**Usage**

```
    INTEGER FUNCTION vlistInqVarTsteptype(INTEGER vlistID, INTEGER 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_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**

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

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

varID     Variable identifier.

code      Code number.

### 4.3.6. Get the Code number of a Variable: vlistInqVarCode

The function vlistInqVarCode returns the code number of a variable.

**Usage**

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

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

varID     Variable identifier.

**Result**

vlistInqVarCode returns the code number of the variable.

### 4.3.7. Define the data type of a Variable: vlistDefVarDatatype

The function vlistDefVarDatatype defines the data type of a variable.

**Usage**

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

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

varID     Variable identifier.

datatype  The data type identifier. The valid **CDI** data types are 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.

### 4.3.8. Get the data type of a Variable: vlistInqVarDatatype

The function vlistInqVarDatatype returns the data type of a variable.

**Usage**

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

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

varID     Variable identifier.

**Result**

vlistInqVarDatatype returns an identifier to the data type of the variable. The valid **CDI** data types are 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.

### 4.3.9. Define the missing value of a Variable: vlistDefVarMissval

The function vlistDefVarMissval defines the missing value of a variable.

**Usage**

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

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

varID     Variable identifier.

missval   Missing value.

### 4.3.10. Get the missing value of a Variable: vlistInqVarMissval

The function vlistInqVarMissval returns the missing value of a variable.

**Usage**

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

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

varID     Variable identifier.

**Result**

vlistInqVarMissval returns the missing value of the variable.

## 4.4. Key attributes

Attributes are metadata used to describe variables or a data set. CDI distinguishes between key attributes and user attributes. User defined attributes are described in the next chapter.

Key attributes are attributes that are interpreted by CDI. An example is the name or the units of a variable.

Key attributes can be defined for data variables and coordinate variables Use the variable ID or one of the following identifiers for the coordinates:

| | |
|---|---|
| CDI_XAXIS | X-axis ID |
| CDI_YAXIS | Y-axis ID |
| CDI_GLOBAL | Global Z-axis |

Some keys like name and units can be used for all variables. Other keys are very special and should only be used for certain variables. The user is also responsible for the data type of the key. CDI supports string, integer, floating point and byte array key attributes. The following key attributes are available:

**String keys**

| | |
|---|---|
| CDI_KEY_NAME | Variable name |
| CDI_KEY_LONGNAME | Long name of the variable |
| CDI_KEY_STDNAME | CF Standard name of the variable |
| CDI_KEY_UNITS | Units of the variable |
| CDI_KEY_REFERENCEURI | Reference URI to grid file |

**Integer keys**

| | |
|---|---|
| CDI_KEY_NUMBEROFGRIDUSED | GRIB2 numberOfGridUsed |
| CDI_KEY_NUMBEROFGRIDINREFERENCE | GRIB2 numberOfGridInReference |
| CDI_KEY_NUMBEROFVGRIDUSED | GRIB2 numberOfVGridUsed |
| CDI_KEY_NLEV | GRIB2 nlev |
| CDI_KEY_CHUNKTYPE | ChunkType: CDI_CHUNK_AUTO/CDI_CHUNK_GRID/CDI_CHUNK |
| CDI_KEY_CHUNKSIZE | ChunkSize |

**Floating point keys**

| | |
|---|---|
| CDI_KEY_MISSVAL | Missing value |

**Byte array keys**

| | |
|---|---|
| CDI_KEY_UUID | UUID for grid/Z-axis reference [size: CDI_UUID_SIZE] |

### 4.4.1. Define a string from a key: `cdiDefKeyString`

The function `cdiDefKeyString` defines a text string from a key.

**Usage**

```
INTEGER FUNCTION cdiDefKeyString(INTEGER cdiID, INTEGER varID, INTEGER key,
                                 CHARACTER*(*) string)
```

`cdiID`   CDI object ID (vlistID, gridID, zaxisID).

`varID`   Variable identifier or CDI_GLOBAL.

`key`     The key to be searched.

`string`  The address of a string where the data will be read.

**Result**

`cdiDefKeyString` returns CDI_NOERR if OK.

**Example**

Here is an example using `cdiDefKeyString` to define the name of a variable:

**INCLUDE** 'cdi.inc'
   ...
**INTEGER** vlistID, varID, **status**
   ...
vlistID  = vlistCreate()
varID = vlistDefVar(vlistID, gridID, zaxisID, TIME_VARYING)
   ...
**status** = cdiDefKeyString(vlistID, varID, CDI_KEY_NAME, "temperature")
   ...

### 4.4.2. Get a string from a key: `cdiInqKeyString`

The function `cdiInqKeyString` gets a text string from a key.

**Usage**

```
INTEGER FUNCTION cdiInqKeyString(INTEGER cdiID, INTEGER varID, INTEGER key,
                                 CHARACTER*(*) string, INTEGER length)
```

`cdiID`   CDI object ID (vlistID, gridID, zaxisID).

`varID`   Variable identifier or CDI_GLOBAL.

`key`     The key to be searched.

`string`  The address of a string where the data will be retrieved. The caller must allocate space for the returned string.

`length`  The allocated length of the string on input.

**Result**

`cdiInqKeyString` returns CDI_NOERR if key is available.

**Example**

Here is an example using `cdiInqKeyString` to get the name of the first variable:

**INCLUDE** 'cdi.inc'
   ...
#define STRLEN 256
   ...
**INTEGER** streamID, vlistID, varID, **status**
**INTEGER** length = STRLEN

**CHARACTER name**[STRLEN]
    ...
streamID = streamOpenRead(...)
vlistID = streamInqVlist(streamID)
    ...
varID = 0
**status** = cdiInqKeyString(vlistID, varID, CDI_KEY_NAME, **name**, length)
    ...

### 4.4.3. Define an integer value from a key: `cdiDefKeyInt`

The function `cdiDefKeyInt` defines an integer value from a key.

**Usage**

```
INTEGER FUNCTION cdiDefKeyInt(INTEGER cdiID, INTEGER varID, INTEGER key,
                             INTEGER value)
```

  `cdiID`   CDI object ID (vlistID, gridID, zaxisID).

  `varID`   Variable identifier or CDI_GLOBAL.

  `key`     The key to be searched.

  `value`  An integer where the data will be read.

**Result**

`cdiDefKeyInt` returns CDI_NOERR if OK.

### 4.4.4. Get an integer value from a key: `cdiInqKeyInt`

The function `cdiInqKeyInt` gets an integer value from a key.

**Usage**

```
INTEGER FUNCTION cdiInqKeyInt(INTEGER cdiID, INTEGER varID, INTEGER key,
                             INTEGER value)
```

  `cdiID`   CDI object ID (vlistID, gridID, zaxisID).

  `varID`   Variable identifier or CDI_GLOBAL.

  `key`     The key to be searched..

  `value`  The address of an integer where the data will be retrieved.

**Result**

`cdiInqKeyInt` returns CDI_NOERR if key is available.

### 4.4.5. Define a floating point value from a key: `cdiDefKeyFloat`

The function `cdiDefKeyFloat` defines a **CDI** floating point value from a key.

**Usage**

```
INTEGER FUNCTION cdiDefKeyFloat(INTEGER cdiID, INTEGER varID, INTEGER key,
                                REAL*8 value)
```

cdiID    CDI object ID (vlistID, gridID, zaxisID).

varID    Variable identifier or CDI_GLOBAL.

key      The key to be searched

value    A double where the data will be read

**Result**

cdiDefKeyFloat returns CDI_NOERR if OK.

### 4.4.6. Get a floating point value from a key: cdiInqKeyFloat

The function cdiInqKeyFloat gets a floating point value from a key.

**Usage**

```
INTEGER FUNCTION cdiInqKeyFloat(INTEGER cdiID, INTEGER varID, INTEGER key,
                                REAL*8 value)
```

cdiID    CDI object ID (vlistID, gridID, zaxisID).

varID    Variable identifier or CDI_GLOBAL.

key      The key to be searched.

value    The address of a double where the data will be retrieved.

**Result**

cdiInqKeyFloat returns CDI_NOERR if key is available.

### 4.4.7. Define a byte array from a key: cdiDefKeyBytes

The function cdiDefKeyBytes defines a byte array from a key.

**Usage**

```
INTEGER FUNCTION cdiDefKeyBytes(INTEGER cdiID, INTEGER varID, INTEGER key,
                                unsigned CHARACTER*(*) bytes, INTEGER length)
```

cdiID    CDI object ID (vlistID, gridID, zaxisID).

varID    Variable identifier or CDI_GLOBAL.

key      The key to be searched.

bytes    The address of a byte array where the data will be read.

length  Length of the byte array

**Result**

cdiDefKeyBytes returns CDI_NOERR if OK.

### 4.4.8. Get a byte array from a key: cdiInqKeyBytes

The function cdiInqKeyBytes gets a byte array from a key.

**Usage**

```
    INTEGER FUNCTION cdiInqKeyBytes(INTEGER cdiID, INTEGER varID, INTEGER key,
                                    unsigned CHARACTER*(*) bytes, INTEGER length)
```

cdiID     CDI object ID (vlistID, gridID, zaxisID).

varID     Variable identifier or CDI_GLOBAL.

key       The key to be searched.

bytes     The address of a byte array where the data will be retrieved. The caller must
          allocate space for the returned byte array.

length    The allocated length of the byte array on input.

**Result**

cdiInqKeyBytes returns CDI_NOERR if key is available.

## 4.5. User attributes

Attributes are metadata used to describe variables or a data set. CDI distinguishes between key attributes and user attributes. Key attributes are described in the last chapter.

User defined attributes are additional attributes that are not interpreted by CDI. These attributes are only available for NetCDF datasets. Here they correspond to all attributes that are not used by CDI as key attributes.

A user defined 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 variables list is associated with a stream.

It is also possible to have attributes that are not associated with any variable. These are called global attributes and are identified by using CDI_GLOBAL as a variable pseudo-ID. Global attributes are usually related to the dataset as a whole.

CDI supports integer, floating point and text attributes. The data types are defined by the following predefined constants:

| | |
|---|---|
| CDI_DATATYPE_INT16 | 16-bit integer attribute |
| CDI_DATATYPE_INT32 | 32-bit integer attribute |
| CDI_DATATYPE_FLT32 | 32-bit floating point attribute |
| CDI_DATATYPE_FLT64 | 64-bit floating point attribute |
| CDI_DATATYPE_TXT | Text attribute |

### 4.5.1. Get number of attributes: cdiInqNatts

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

**Usage**

    INTEGER FUNCTION cdiInqNatts(INTEGER cdiID, INTEGER varID, INTEGER 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.

### 4.5.2. Get information about an attribute: cdiInqAtt

The function cdiInqAtt gets information about an attribute.

**Usage**

    INTEGER FUNCTION cdiInqAtt(INTEGER cdiID, INTEGER varID, INTEGER attnum,
                              CHARACTER*(*) name, INTEGER typep, INTEGER 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.

### 4.5.3. Define a text attribute: `cdiDefAttTxt`

The function `cdiDefAttTxt` defines a text attribute.

**Usage**

```
INTEGER FUNCTION cdiDefAttTxt(INTEGER cdiID, INTEGER varID, CHARACTER*(*) name,
                             INTEGER len, CHARACTER*(*) tp)
```

cdiID   CDI ID, from a previous call to vlistCreate, gridCreate or zaxisCreate.

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.

**Example**

Here is an example using `cdiDefAttTxt` to define the attribute "description":

> **INCLUDE** 'cdi.inc'
>     ...
> **INTEGER** vlistID, varID, **status**
> **CHARACTER** text[] = "description-of-the-variable"
>     ...
> vlistID  = vlistCreate()
> varID = vlistDefVar(vlistID, gridID, zaxisID, TIME_VARYING)
>     ...
> **status** = cdiDefAttTxt(vlistID, varID, "description", **LEN**(text), text)
>     ...

### 4.5.4. Get the value(s) of a text attribute: `cdiInqAttTxt`

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

**Usage**

```
INTEGER FUNCTION cdiInqAttTxt(INTEGER cdiID, INTEGER varID, CHARACTER*(*) name,
                             INTEGER mlen, CHARACTER*(*) tp)
```

cdiID   CDI ID, from a previous call to vlistCreate, gridCreate or zaxisCreate.

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).

### 4.5.5. Define an integer attribute: `cdiDefAttInt`

The function `cdiDefAttInt` defines an integer attribute.

**Usage**

```
INTEGER FUNCTION cdiDefAttInt(INTEGER cdiID, INTEGER varID, CHARACTER*(*) name,
                             INTEGER type, INTEGER len, INTEGER ip)
```

cdiID    CDI ID, from a previous call to vlistCreate, gridCreate or zaxisCreate.

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.

### 4.5.6. Get the value(s) of an integer attribute: cdiInqAttInt

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

**Usage**

```
INTEGER FUNCTION cdiInqAttInt(INTEGER cdiID, INTEGER varID, CHARACTER*(*) name,
                             INTEGER mlen, INTEGER ip)
```

cdiID    CDI ID, from a previous call to vlistCreate, gridCreate or zaxisCreate.

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).

### 4.5.7. Define a floating point attribute: cdiDefAttFlt

The function cdiDefAttFlt defines a floating point attribute.

**Usage**

```
INTEGER FUNCTION cdiDefAttFlt(INTEGER cdiID, INTEGER varID, CHARACTER*(*) name,
                             INTEGER type, INTEGER len, REAL*8 dp)
```

cdiID    CDI ID, from a previous call to vlistCreate, gridCreate or zaxisCreate.

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.

### 4.5.8. Get the value(s) of a floating point attribute: cdiInqAttFlt

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

**Usage**

```
INTEGER FUNCTION cdiInqAttFlt(INTEGER cdiID, INTEGER varID, CHARACTER*(*) name,
                             INTEGER mlen, REAL*8 dp)
```

    `cdiID`   CDI ID, from a previous call to `vlistCreate`, `gridCreate` or `zaxisCreate`.

    `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).

## 4.6. 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_PROJECTION | Projected coordinates |
| GRID_SPECTRAL | Spherical harmonic coefficients |
| GRID_GME | Icosahedral-hexagonal GME grid |
| GRID_CURVILINEAR | Curvilinear grid |
| GRID_UNSTRUCTURED | Unstructured grid |

### 4.6.1. Create a horizontal Grid: `gridCreate`

The function `gridCreate` creates a horizontal Grid.

**Usage**

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

gridtype    The type of the grid, one of the set of predefined **CDI** grid types. The valid **CDI** grid types are GRID_GENERIC, GRID_LONLAT, GRID_GAUSSIAN, 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.inc'
    ...
#define  nlon  12
#define  nlat   6
    ...
REAL*8 lons(nlon) = (/0, 30, 60, 90, 120, 150, 180, 210, 240, 270, 300, 330/)
REAL*8 lats(nlat) = (/-75, -45, -15, 15, 45, 75/)
INTEGER gridID
    ...
gridID = gridCreate(GRID_LONLAT, nlon*nlat)
CALL gridDefXsize(gridID, nlon)
CALL gridDefYsize(gridID, nlat)
CALL gridDefXvals(gridID, lons)
CALL gridDefYvals(gridID, lats)
    ...
```

### 4.6.2. Destroy a horizontal Grid: gridDestroy

**Usage**

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

gridID   Grid ID, from a previous call to [gridCreate](gridCreate).

### 4.6.3. Duplicate a horizontal Grid: gridDuplicate

The function gridDuplicate duplicates a horizontal Grid.

**Usage**

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

gridID   Grid ID, from a previous call to [gridCreate](gridCreate) or [vlistInqVarGrid](vlistInqVarGrid).

**Result**

gridDuplicate returns an identifier to the duplicated Grid.

### 4.6.4. Get the type of a Grid: gridInqType

The function gridInqType returns the type of a Grid.

**Usage**

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

gridID   Grid ID, from a previous call to [gridCreate](gridCreate) or [vlistInqVarGrid](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_LONLAT, GRID_GAUSSIAN, GRID_PROJECTION, GRID_SPECTRAL, GRID_GME, GRID_CURVILINEAR and GRID_UNSTRUCTURED.

### 4.6.5. Get the size of a Grid: gridInqSize

The function gridInqSize returns the size of a Grid.

**Usage**

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

gridID   Grid ID, from a previous call to [gridCreate](gridCreate) or [vlistInqVarGrid](vlistInqVarGrid).

**Result**

gridInqSize returns the number of grid points of a Grid.

### 4.6.6. Define the number of values of a X-axis: gridDefXsize

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

**Usage**

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

gridID    Grid ID, from a previous call to gridCreate.

xsize    Number of values of a X-axis.

### 4.6.7. Get the number of values of a X-axis: gridInqXsize

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

**Usage**

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

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

**Result**

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

### 4.6.8. Define the number of values of a Y-axis: gridDefYsize

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

**Usage**

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

gridID    Grid ID, from a previous call to gridCreate.

ysize    Number of values of a Y-axis.

### 4.6.9. Get the number of values of a Y-axis: gridInqYsize

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

**Usage**

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

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

**Result**

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

### 4.6.10. Define the number of parallels between a pole and the equator: gridDefNP

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

**Usage**

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

gridID    Grid ID, from a previous call to gridCreate.

np    Number of parallels between a pole and the equator.

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

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

**Usage**

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

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

**Result**

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

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

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

**Usage**

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

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

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

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

**Usage**

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

gridID   Grid ID, from a previous call to `gridCreate` or `vlistInqVarGrid`.
xvals    Pointer to the location into which the X-values are read. The caller must allocate space for the returned values.

**Result**

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

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

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

**Usage**

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

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

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

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

**Usage**

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

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

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

**Result**

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

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

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

**Usage**

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

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

xbounds X-bounds of the grid.

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

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

**Usage**

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

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

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

**Result**

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

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

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

**Usage**

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

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

ybounds Y-bounds of the grid.

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

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

**Usage**

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

gridID    Grid ID, from a previous call to [gridCreate](#) or [vlistInqVarGrid](#).

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

**Result**

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

## 4.7. 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 zaxis type |
| ZAXIS_SURFACE | Surface level |
| ZAXIS_HYBRID | Hybrid level |
| ZAXIS_SIGMA | Sigma level |
| ZAXIS_PRESSURE | Isobaric pressure level in Pascal |
| ZAXIS_HEIGHT | Height above ground in meters |
| ZAXIS_ISENTROPIC | Isentropic (theta) level |
| ZAXIS_ALTITUDE | Altitude above mean sea level in meters |
| ZAXIS_MEANSEA | Mean sea level |
| ZAXIS_TOA | Norminal top of atmosphere |
| ZAXIS_SEA_BOTTOM | Sea bottom |
| ZAXIS_ATMOSPHERE | Entire atmosphere |
| 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 |

### 4.7.1. Create a vertical Z-axis: `zaxisCreate`

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

**Usage**

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

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

size       Number of levels.

**Result**

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

**Example**

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

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

### 4.7.2. Destroy a vertical Z-axis: `zaxisDestroy`

**Usage**

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

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

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

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

**Usage**

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

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

**Result**

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

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

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

**Usage**

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

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

**Result**

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

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

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

**Usage**

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

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

levels   All levels of the Z-axis.

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

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

**Usage**

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

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

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

**Result**

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

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

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

**Usage**

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

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

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

**Result**

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

## 4.8. T-axis functions

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

| | |
|---|---|
| TAXIS_ABSOLUTE | Absolute time axis |
| TAXIS_RELATIVE | Relative time axis |

An absolute time axis has the current time to each time step. It can be used without knowledge of the calendar.

A relative time is the time relative to a fixed reference time. The current time results from the reference time and the elapsed interval. The result depends on the used calendar. CDI supports the following calendar types:

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

### 4.8.1. Create a Time axis: taxisCreate

The function taxisCreate creates a Time axis.

**Usage**

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

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

**Result**

taxisCreate returns an identifier to the Time axis.

**Example**

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

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

### 4.8.2. **Destroy a Time axis:** `taxisDestroy`

**Usage**

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

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

### 4.8.3. **Define the reference date:** `taxisDefRdate`

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

**Usage**

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

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

### 4.8.4. **Get the reference date:** `taxisInqRdate`

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

**Usage**

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

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

**Result**

`taxisInqRdate` returns the reference date.

### 4.8.5. **Define the reference time:** `taxisDefRtime`

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

**Usage**

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

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

### 4.8.6. **Get the reference time:** `taxisInqRtime`

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

**Usage**

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

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

**Result**

`taxisInqRtime` returns the reference time.

### 4.8.7. Define the verification date: `taxisDefVdate`

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

**Usage**

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

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

### 4.8.8. Get the verification date: `taxisInqVdate`

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

**Usage**

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

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

**Result**

`taxisInqVdate` returns the verification date.

### 4.8.9. Define the verification time: `taxisDefVtime`

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

**Usage**

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

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

### 4.8.10. Get the verification time: `taxisInqVtime`

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

**Usage**

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

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

**Result**

`taxisInqVtime` returns the verification time.

### 4.8.11. Define the calendar: `taxisDefCalendar`

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

**Usage**

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

taxisID    Time axis ID, from a previous call to taxisCreate

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

### 4.8.12. Get the calendar: taxisInqCalendar

The function taxisInqCalendar returns the calendar of a Time axis.

**Usage**

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

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

**Result**

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

# Bibliography

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

### cdiDefAttFlt

```
INTEGER FUNCTION cdiDefAttFlt(INTEGER cdiID, INTEGER varID, CHARACTER*(*) name,
                              INTEGER type, INTEGER len, REAL*8 dp)
```

Define a floating point attribute

### cdiDefAttInt

```
INTEGER FUNCTION cdiDefAttInt(INTEGER cdiID, INTEGER varID, CHARACTER*(*) name,
                              INTEGER type, INTEGER len, INTEGER ip)
```

Define an integer attribute

### cdiDefAttTxt

```
INTEGER FUNCTION cdiDefAttTxt(INTEGER cdiID, INTEGER varID, CHARACTER*(*) name,
                              INTEGER len, CHARACTER*(*) tp)
```

Define a text attribute

### cdiDefKeyBytes

```
INTEGER FUNCTION cdiDefKeyBytes(INTEGER cdiID, INTEGER varID, INTEGER key,
                                unsigned CHARACTER*(*) bytes, INTEGER length)
```

Define a byte array from a key

### cdiDefKeyFloat

```
INTEGER FUNCTION cdiDefKeyFloat(INTEGER cdiID, INTEGER varID, INTEGER key,
                                REAL*8 value)
```

Define a floating point value from a key

### cdiDefKeyInt

```
INTEGER FUNCTION cdiDefKeyInt(INTEGER cdiID, INTEGER varID, INTEGER key,
                              INTEGER value)
```

Define an integer value from a key

## cdiDefKeyString

```
INTEGER FUNCTION cdiDefKeyString(INTEGER cdiID, INTEGER varID, INTEGER key,
                                 CHARACTER*(*) string)
```

Define a string from a key

## cdiInqAtt

```
INTEGER FUNCTION cdiInqAtt(INTEGER cdiID, INTEGER varID, INTEGER attnum,
                           CHARACTER*(*) name, INTEGER typep, INTEGER lenp)
```

Get information about an attribute

## cdiInqAttFlt

```
INTEGER FUNCTION cdiInqAttFlt(INTEGER cdiID, INTEGER varID, CHARACTER*(*) name,
                              INTEGER mlen, REAL*8 dp)
```

Get the value(s) of a floating point attribute

## cdiInqAttInt

```
INTEGER FUNCTION cdiInqAttInt(INTEGER cdiID, INTEGER varID, CHARACTER*(*) name,
                              INTEGER mlen, INTEGER ip)
```

Get the value(s) of an integer attribute

## cdiInqAttTxt

```
INTEGER FUNCTION cdiInqAttTxt(INTEGER cdiID, INTEGER varID, CHARACTER*(*) name,
                              INTEGER mlen, CHARACTER*(*) tp)
```

Get the value(s) of a text attribute

## cdiInqKeyBytes

```
INTEGER FUNCTION cdiInqKeyBytes(INTEGER cdiID, INTEGER varID, INTEGER key,
                                unsigned CHARACTER*(*) bytes, INTEGER length)
```

Get a byte array from a key

## cdiInqKeyFloat

```
INTEGER FUNCTION cdiInqKeyFloat(INTEGER cdiID, INTEGER varID, INTEGER key,
                                REAL*8 value)
```

Get a floating point value from a key

### cdiInqKeyInt

```
INTEGER FUNCTION cdiInqKeyInt(INTEGER cdiID, INTEGER varID, INTEGER key,
                             INTEGER value)
```

Get an integer value from a key

### cdiInqKeyString

```
INTEGER FUNCTION cdiInqKeyString(INTEGER cdiID, INTEGER varID, INTEGER key,
                                CHARACTER*(*) string, INTEGER length)
```

Get a string from a key

### cdiInqNatts

```
INTEGER FUNCTION cdiInqNatts(INTEGER cdiID, INTEGER varID, INTEGER nattsp)
```

Get number of attributes

### gridCreate

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

Create a horizontal Grid

### gridDefNP

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

Define the number of parallels between a pole and the equator

### gridDefXbounds

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

Define the bounds of a X-axis

### gridDefXsize

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

Define the number of values of a X-axis

### gridDefXvals

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

Define the values of a X-axis

## gridDefYbounds

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

Define the bounds of a Y-axis

## gridDefYsize

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

Define the number of values of a Y-axis

## gridDefYvals

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

Define the values of a Y-axis

## gridDestroy

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

Destroy a horizontal Grid

## gridDuplicate

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

Duplicate a horizontal Grid

## gridInqNP

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

Get the number of parallels between a pole and the equator

## gridInqSize

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

Get the size of a Grid

## gridInqType

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

Get the type of a Grid

### gridInqXbounds

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

Get the bounds of a X-axis

### gridInqXsize

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

Get the number of values of a X-axis

### gridInqXvals

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

Get all values of a X-axis

### gridInqYbounds

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

Get the bounds of a Y-axis

### gridInqYsize

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

Get the number of values of a Y-axis

### gridInqYvals

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

Get all values of a Y-axis

### streamClose

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

Close an open dataset

### streamDefByteorder

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

Define the byte order

## streamDefRecord

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

Define the next record

## streamDefTimestep

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

Define a timestep

## streamDefVlist

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

Define the variable list

## streamInqByteorder

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

Get the byte order

## streamInqFiletype

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

Get the filetype

## streamInqTimestep

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

Get timestep information

## streamInqVlist

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

Get the variable list

## streamOpenRead

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

Open a dataset for reading

### streamOpenWrite

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

Create a new dataset

### streamReadVar

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

Read a variable

### streamReadVarF

```
SUBROUTINE streamReadVar(INTEGER streamID, INTEGER varID, REAL*4 data,
                         INTEGER numMissVals)
```

Read a variable

### streamReadVarSlice

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

Read a horizontal slice of a variable

### streamReadVarSliceF

```
SUBROUTINE streamReadVarSliceF(INTEGER streamID, INTEGER varID, INTEGER levelID,
                               REAL*4 data, INTEGER numMissVals)
```

Read a horizontal slice of a variable

### streamWriteVar

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

Write a variable

### streamWriteVarF

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

Write a variable

## streamWriteVarSlice

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

Write a horizontal slice of a variable

## streamWriteVarSliceF

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

Write a horizontal slice of a variable

## taxisCreate

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

Create a Time axis

## taxisDefCalendar

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

Define the calendar

## taxisDefRdate

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

Define the reference date

## taxisDefRtime

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

Define the reference time

## taxisDefVdate

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

Define the verification date

## taxisDefVtime

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

Define the verification time

### taxisDestroy

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

Destroy a Time axis

### taxisInqCalendar

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

Get the calendar

### taxisInqRdate

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

Get the reference date

### taxisInqRtime

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

Get the reference time

### taxisInqVdate

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

Get the verification date

### taxisInqVtime

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

Get the verification time

### vlistCat

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

Concatenate two variable lists

### vlistCopy

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

Copy a variable list

## vlistCopyFlag

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

Copy some entries of a variable list

## vlistCreate

```
INTEGER FUNCTION vlistCreate()
```

Create a variable list

## vlistDefTaxis

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

Define the time axis

## vlistDefVar

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

Define a Variable

## vlistDefVarCode

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

Define the code number of a Variable

## vlistDefVarDatatype

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

Define the data type of a Variable

## vlistDefVarMissval

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

Define the missing value of a Variable

## vlistDestroy

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

Destroy a variable list

## vlistDuplicate

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

Duplicate a variable list

## vlistInqTaxis

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

Get the time axis

## vlistInqVarCode

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

Get the Code number of a Variable

## vlistInqVarDatatype

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

Get the data type of a Variable

## vlistInqVarGrid

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

Get the Grid ID of a Variable

## vlistInqVarMissval

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

Get the missing value of a Variable

## vlistInqVarTsteptype

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

Get the timestep type of a Variable

## vlistInqVarZaxis

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

Get the Zaxis ID of a Variable

## vlistNumGrids

    INTEGER FUNCTION vlistNumGrids(INTEGER vlistID)

Number of grids in a variable list

## vlistNumZaxis

    INTEGER FUNCTION vlistNumZaxis(INTEGER vlistID)

Number of zaxis in a variable list

## vlistNvars

    INTEGER FUNCTION vlistNvars(INTEGER vlistID)

Number of variables in a variable list

## zaxisCreate

    INTEGER FUNCTION zaxisCreate(INTEGER zaxistype, INTEGER size)

Create a vertical Z-axis

## zaxisDefLevels

    SUBROUTINE zaxisDefLevels(INTEGER zaxisID, REAL*8 levels)

Define the levels of a Z-axis

## zaxisDestroy

    SUBROUTINE zaxisDestroy(INTEGER zaxisID)

Destroy a vertical Z-axis

## zaxisInqLevel

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

Get one level of a Z-axis

## zaxisInqLevels

    SUBROUTINE zaxisInqLevels(INTEGER zaxisID, REAL*8 levels)

Get all levels of a Z-axis

## zaxisInqSize

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

Get the size of a Z-axis

## zaxisInqType

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

Get the type of a Z-axis

# 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.

```fortran
      PROGRAM CDIWRITE

      IMPLICIT NONE

      INCLUDE 'cdi.inc'

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

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

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

      nmiss = 0

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

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

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

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

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

      !       Define the variable names
              CALL vlistDefVarName(vlistID, varID1, "varname1")
              CALL vlistDefVarName(vlistID, varID2, "varname2")
50
      !       Create a Time axis
              taxisID = taxisCreate(TAXIS_ABSOLUTE)

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

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

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

      !       Loop over the number of time steps
              DO tsID = 0, nts-1
      !         Set the verification date to 1985-01-01 + tsID
70              CALL taxisDefVdate(taxisID, 19850101+tsID)
      !         Set the verification time to 12:00:00
                CALL taxisDefVtime(taxisID, 120000)
      !         Define the time step
                status = streamDefTimestep(streamID, tsID)
75
      !         Init var1 and var2
                DO i = 1, nlon*nlat
                  var1(i) = 1.1
                END DO
80              DO i = 1, nlon*nlat*nlev
                  var2(i) = 2.2
                END DO

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

      !       Close the output stream
90            CALL streamClose(streamID)

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

```
      END
```

### B.1.1. Result

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

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

    lon = 0, 30, 60, 90, 120, 150, 180, 210, 240, 270, 300, 330 ;
26
    lat = −75, −45, −15, 15, 45, 75 ;

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

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

## B.2. Read a dataset

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

```
      PROGRAM CDIREAD

3     IMPLICIT NONE

      INCLUDE 'cdi.inc'

      INTEGER nlon, nlat
8     INTEGER nlev, nts
      PARAMETER (nlon = 12) ! Number of longitudes
      PARAMETER (nlat = 6) ! Number of latitudes
      PARAMETER (nlev = 5) ! Number of levels
      PARAMETER (nts = 3) ! Number of time steps
13
```

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

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

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

!       Set the  variable IDs
        varID1 = 0
        varID2 = 1
33
!       Get the Time axis from the variable  list
        taxisID = vlistInqTaxis( vlistID )

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

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


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

53 !    Close the input stream
        CALL streamClose(streamID)

        END
```

## B.3. Copy a dataset

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

```fortran
        PROGRAM CDICOPY

        IMPLICIT NONE
4
        INCLUDE 'cdi.inc'

        INTEGER nlon, nlat, nlev, nts
```

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

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

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


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

29 !    Set the variable  IDs
      varID1 = 0
      varID2 = 1


   !    Open the output dataset (GRIB format)
34    streamID2 = streamOpenWrite("example.grb", CDI_FILETYPE_GRB)
      IF ( streamID2 < 0 ) THEN
         WRITE(0,*) cdiStringError(streamID2)
         STOP
      END IF
39
      vlistID2  = vlistDuplicate( vlistID1 )

      CALL streamDefVlist(streamID2, vlistID2)


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


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


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


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


   !    Close the streams
      CALL streamClose(streamID1)
      CALL streamClose(streamID2)
64
```

```
        END
```

## B.4. Fortran 2003: mo_cdi and iso_c_binding

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

- appending '\0' with `//C_NULL_CHAR`

- prepending `C_CHAR_` to plain charcters

- take ctrim from `mo_cdi` for `CHARACTER(type=c_char)` variables

```
     PROGRAM CDIREADF2003
        use iso_c_binding
        use mo_cdi

 5      IMPLICIT NONE

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

        cdi_version => cdiLibraryVersion()
20
        WRITE (0, '(a,132a)') 'cdi-version:', cdi_version

        ! Open the dataset
        streamID = streamOpenRead(C_CHAR_"example_f2003.nc"//C_NULL_CHAR)
25      IF ( streamID < 0 ) THEN
          PRINT *, 'Could not Read the file.'
          msg => cdiStringError(streamID)
          WRITE(0,'(132a)') msg
          STOP 1
30      END IF

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

35      nvars = vlistNvars(vlistID)

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

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

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

   END DO
55
   ! Get the Time axis form the variable  list
   taxisID = vlistInqTaxis( vlistID )

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

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

     PRINT '(a,i3,i10,i10)', 'Timestep:', tsID+1, vdate, vtime
70
     ! Read the variables  at the current timestep
     DO varID = 0, nvars−1
       gridID = vlistInqVarGrid(vlistID,  varID)
       gsize  = gridInqSize(gridID)
75     zaxisID = vlistInqVarZaxis(vlistID,  varID)
       nlevel  = zaxisInqSize(zaxisID)
       ALLOCATE(field(gsize, nlevel))
       CALL streamReadVar(streamID, varID, field, nmiss)
       DO ilev = 1, nlevel
80       PRINT '(a,i3,a,i3,a,f10.5,1x,f10.5)',  '   var=', varID+1, &
              ' level=',  ilev ,  ':',  &
              MINVAL(field(:,ilev)), MAXVAL(field(:,ilev))
       END DO
       DEALLOCATE(field)
85   END DO
   END DO

   ! Close the input stream
   CALL streamClose(streamID)
90
END PROGRAM CDIREADF2003
```

```fortran
      PROGRAM CDIWRITEF2003

      USE iso_c_binding
4     USE mo_cdi

      IMPLICIT NONE
```

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

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

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

      nmiss = 0

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

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

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

!     Create a variable list
      vlistID = vlistCreate()

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

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

!     Create a Time axis
      taxisID = taxisCreate(TAXIS_ABSOLUTE)

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

!     Create a dataset in netCDF format
      streamID = streamOpenWrite(C_CHAR_"example_f2003.nc"//C_NULL_CHAR, CDI_FILETYPE_NC)
      IF ( streamID < 0 ) THEN
```

```
64          msg => cdiStringError(streamID)
            WRITE(0,'(132a)') msg
            STOP 1
         END IF

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

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

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

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

94  !      Close the output stream
         CALL streamClose(streamID)

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

104      END PROGRAM CDIWRITEF2003
```

# C. Environment Variables

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

| Variable name | Default | Description |
|---|---|---|
| CDI_CONVERT_CUBESPHERE | 1 | Convert cubed-sphere data to unstructured grid. |
| CDI_CHUNK_CACHE | 0 | Set the NetCDF4 chunk cache size. |
| CDI_CHUNK_CACHE_MAX | 0 | Set maximum chunk cache size. |
| CDI_GRIB1_TEMPLATE | None | Path to a GRIB1 template file for GRIB_API. |
| CDI_GRIB2_TEMPLATE | None | Path to a GRIB2 template file for GRIB_API. |
| CDI_INVENTORY_MODE | None | Set to time to skip double variable entries. |
| CDI_QUERY_ABORT | 1 | Abort if query entry not found |
| CDI_READ_CELL_CORNERS | 1 | Read grid cell corners. |
| CDI_SHUFFLE | 0 | Set shuffle option to NetCDF4 deflation compression. |
| CDI_VERSION_INFO | 1 | Set to 0 to disable NetCDF global attribute CDI. |

# Index

## Z