Technical report; TR-224

KNMI

HDFg library and some HDF utilities

an extention to the NCSA HDF library

user's manual & reference guide

Han The

De Bilt, 1999

Koninklijk Nederlands Meteorologisch Instituut

Technical report = Technisch rapport; TR-224

De Bilt, 1999

P.O. Box 201
3730 AE De Bilt
Wilhelminalaan 10
Telephone +31 30 220 69 11
Telefax +31 30 221 04 07

Author: Han The

UDC: 681.3.06
551.50
(03)

ISSN: 0169-1708

ISBN: 90-369-2168-6

HDFg library and some HDF utilities

An Extension to the NCSA HDF Library

User's Manual & Reference Guide

Han The, October 1999

This report describes the HDFg library. HDFg comprises a set of high-level HDF I/O functions, built on top of the official NCSA HDF library (version 4.1). They were developed to simplify I/O from and to HDF scientific data sets (i.e. multi-dimensional arrays) and their attributes. Furthermore as set of utilities were developed to convert the data from asimof, ASCII or plain binary format to HDF. This report gives a full description of the HDF attributes used to interpret the GRIB header information. These conventions are used by the `asim2hdf` utility. The HDFg library was developed as part of the WEPTEL project.

How to obtain the HDF library?

The HDF library can be downloaded from the internet address:

```
http://hdf.ncsa.uiuc.edu/
```

or:

```
ftp: hdf.ncsa.uiuc.edu pub/dist/HDF/
```

You can download the source code as well as compiled versions. Be sure to download version 4 (latest release 4.1r3). Additional documentation can also be downloaded. Both the user's guide as well as the reference manual are available as an on-line html document (`http://hdf.ncsa.uiuc.edu/training/-HDFtraining/`) or as a printable postscript file. The chapters in these guides that can be read as an introduction to HDF are "Fundamentals" and "SD API". Items not explained in this report can be found in these manuals. The functions described in this document are available from library `libHDFg1.0`.

How to use the HDF library?

The NCSA HDF-code comprises the following libraries:

```
libmfhdf.a libdf.a libjpeg.a libz.a
```

You should give the following library specifications to compile an application containing `HDFg` functions :

```
-I<path of HDF include directory> -I<path of hdfgrib.h> \
-L<path of HDF libraries> \
<path>/libHDFg1.0.a -lmfhdf -ldf -lz -lm
```

`-lm` refers to the mathematical library and is required for C only. If this does not work, try `-lM`.

Include `HDFg.h` for C and `HDFg.inc` as well as `hdf.inc` for Fortran. `hdf.inc` is an NCSA include-file. `HDFg.h` contains references to the required NCSA HDF includes, which do not need to be included explicitly in addition to `HDFg.h`. If you want to have access to the GRIB data set names (see Appendix 1), then you should include `gribdef.h` as well.
When using Fortran, the function names might need slight (compiler-dependent) modifications.

Type Definitions for Scientific Data Sets (SDS)

Scientific data sets are defined by an array description and the data itself. We combined these to a single structure (C only):

```
typedef struct {
  int32  rank;  /* number of dimensions */
  int32  *dim;
  int32  type;  /* number type (see Table 1) */
  float  *data;
```

```
} SDS;
```

Data sets can be identified by name in an HDF file. We did not include this as part of the structure, because it is auxiliary information not needed to interpret them correctly in terms of computer data.
Typecast `*data` to the appropriate type if `SDS.type` is not equal to `DFNT_FLOAT32`. For example, use `((unsigned short *)sds.data)[i]` to access element `i` in the array if the data represent an array of unsigned shorts. You may also use `(uint16 *)` instead. In context:

```
switch (sds.type) {
  case DFNT_UINT16:
    ..do something with
      ((unsigned short *)sds.data)[i] or ((uint16 *) sds.data)[i]
  break;
  case DFNT_FLOAT32:
    ..do something with sds.data[i]
  break;
}
```

The number-type definitions in Table 1 can be accessed by including `HDFg.h`. The number of bytes occupied by a single value of this type can be obtained by calling `DFKTNsize()`, e.g.:

```
nb = DFKNTsize(DFNT_FLOAT32)  /* nb equals 4 */
```

Table 1. HDF type definitions (NCSA)

DFNT_CHAR8	4	DFNT_UCHAR	3
DFNT_INT8	20	DFNT_UINT8	21
DFNT_INT16	22	DFNT_UINT16	23
DFNT_INT32	24	DFNT_UINT32	25
DFNT_FLOAT32	5	DFNT_FLOAT64	6

Attributes, i.e. auxiliary information also referred to as metadata, are defined by a keyword (name), its number type, length and content. We have combined these to a single structure (C only). For convenience we limited the keyword length to 32 characters. The attribute structure contains a pointer to the next attribute, used to create a linked list when.

```
struct _attribute {
  char  keyword[32];
  int32 len;    /* length of value */
  int32 type;   /* see Table 1 */
  char  *value;
  struct _attribute *next;
};

typedef struct _attribute attribute;
```

If the attribute represents a text string, then `len` must equal the string length *including* the delimiter `'\0'`. Attributes can be stored in an HDF file linked to a data set, or an axis of the data set or just as global information. Note that although attributes can be accessed by names similarly to data sets, we included the name as part of the structure to be able to recognise them in a linked list.

Converting data sets from asimof-format to HDF-format

Data files in asimof format can be converted to HDF format using the utility `asim2hdf`. `asim2hdf` translates the GRIB headers into HDF attributes. Furthermore, data sets in asimof format representing layers of a 3D data set are combined into a single 3D HDF data set. It is assumed that the GRIB headers contain a grid description section (GDS), otherwise conversion will fail. `asim2hdf` cannot be used if the grid has been described implictly (PDS octet 7).

Usage (version 1.1):

```
asim2hdf [-v][-MAXBUF][-f][-{ecmwf,metcast}][-l[fname]] asimof_file hdf_file
```

`-v` stands for verbose;

`-MAXBUF` is used to increase the buffer required containing the data. By default, `MAXBUF` equals 50000 (floats). If you want to convert asimof files containing data sets larger than 50000 floats, you can indicate this by replacing `MAXBUF` with the size required;

`-f` is used to indicate that the data must be stored as floats and not as scaled data;

`-{ecmwf,metcast}`
`asim2hdf` recognises the following local code tables for Section 1 (PDS): ECMWF (version number 128), METCAST (version number 254), and HIRLAM (version number 1). The default conversion is Hirlam (see Appendix 1 for the corresponding HDF data set names).

`-l` convert the data sets read from `stdin` (Use `<cntl>D` to mark end of `stdin` when using this option in interactive mode). Each line is interpreted as a single data-set name. The data set names typed in should be exactly as they would appear in the HDF file (see naming conventions below).

`-lfname` same as `-l`, except that the names are read from file. This option is the same as using `-l` only plus a redirection, i.e. '`asim2hdf -l ..<fin`' instead of '`asim2hdf -lfin ..`'.

Naming Conventions and Attribute Definitions for `asim2hdf`

Data Set Names

The meteorological data set names are stored in `char grib_param[]` (Appendix 1). They correspond to the GRIB definitions in Table 2 in the WMO GRIB reference manual. All data set names are less than 36 characters (including `'\0'`). They do not contain blanks and are written in undercast.
Data set names representing values at the standard heights 2m and 10m above ground are tagged by the extension `_2`, resp. `_10`. Names representing values at mean sea level are tagged by the extension: `_msl`. Data set containing modelling layers get the extension _=.
A data-set name can be referred to as `grib_param[i]`, where `i` is the GRIB data-set code. Also, a list of IDs is available, referring to the elements in `grib_param[]`. e.g.:

```
grib_param[PRESSURE] = "pressure"
```

Therefore, instead of:

```
if (!strcmp(grib_param[i],"pressure")) ...
```

you could write:

```
if (i == PRESSURE) ...
```

These IDs are defined in `gribdef.h`.

Attributes

Attributes are meta-data attached to a file, a data set, or a data-set dimension. The dimension sizes, rank and number type are not considered as attributes and are stored as an intrinsic part of the data set. Attribute names are not fixed. However, a number of names are proposed by the NCSA[1]. We adopted these conventions whenever suitable. The attributes used by `asim2hdf` are listed below. Non-standard attribute names are written in italics.

Note: all names are case-sensitive.

[1] The NCSA recommends a set of standard attribute names. These are: `long_name` (additional name for the array), `units`, `format` (format for dispaying the numerical values), `cordsys`, `valid_range`, `_FillValue`, `scale_factor`, `scale_factor_err`, `add_offset`, `add_offset_err`, `calibrated_nt` (number type of the calibrated data). Unfortunately, even though these attribute names are unique, their contents might be ambiguous, e.g. `cordsys="geographic"` or `cordsys="Geographic"`, referring to the same coordinate system.

object type	attribute	type	count	description
global	*source*	DFNT_CHAR8	*	source from which the data were made available
global or SDS array	*ref_time*	DFNT_FLOAT64	1	time label for the data set contents: yyyymmddhhhh
	cordsys	DFNT_CHAR8	*	coordinate system used for the SDS array
	cordsys_param	DFNT_FLOAT32	*	array of parameters describing cordsys (number of parameters and meaning depend on the type of projection)
SDS array only	unit	DFNT_CHAR8	*	units used for the contents of the data
	scale_factor	DFNT_FLOAT64	1	value by which each array value is to be multiplied
	add_offset	DFNT_FLOAT64	1	value to which each array value is to be added
	precision	DFNT_INT8	1	data precision expressed in bits
	n_scale_factor	DFNT_FLOAT64	*	values by which each value in a plane of a 3D data set is to be multiplied
	n_add_offset	DFNT_FLOAT64	*	values by which each value in a plane of a 3D data set is to be added
	n_precision	DFNT_INT8	*	data precision expressed in bits per plane
	ttype	DFNT_CHAR8	*	time range indicator
	trange	DFNT_FLOAT64	2	time range indicated by ttype (yyyymmddhhhh)
	zlevel	DFNT_CHAR8	*	vertical extent (for data sets 2D in space only)
	zrange	DFNT_FLOAT32	2	vertical range for zlevel
	zunit	DFNT_CHAR8	*	units in which zrange is given
x and y dimension	*step*	DFNT_FLOAT32	1	grid size in degrees or coordinate-system units
	start_value	DFNT_FLOAT32	1	coordinate of first grid point
z dimension	*range*	DFNT_FLOAT32	2	range within the projection space (edges of the grid or position of the first/last grid point)
	unit	DFNT_CHAR8	*	units in which scale and range are expressed
	long_name	DFNT_CHAR8	*	indicator for contents of vertical extent (corresponds to zlevel)

Specifications

Below you find a survey of the attribute contents. The equivalent position in the GRIB header is indicated between brackets. PDS stands for product definition section; i.e. section 1 of the GRIB header. GDS stands for grid description section; i.e. section 2 of the GRIB header.

source — "ecmwf" (98), "knmi" (99), "hirlam" (96), "noaa" (59), "esa" (97), "dmu" (150)
(PDS octet 5) If the station ID differs from the ones indicated, then source contains the ID as a text string;

ref_time — (PDS octets 13-17+25) Reference time. The time is given as a single value: yyyymmddhhhh. e.g.: 199709011230. If the asimof file contains data sets with different reference times, then ref_time is given as a data sets attribute, otherwise ref_time is given as a global attribute;

cordsys — "geographical", "polar_stereographic", "space_view", "shifted_pole"
(GDS octet 6) The remaining projection types have not been implemented. If the asimof file contains data sets with different coordinate systems, then cordsys is given as data set attribute, otherwise cordsys is given as a global attribute;

cordsys_param
These parameters are needed to interpret the geographical coordinates of the projected grid points. If the asimof file contains data sets using various coordinate systems, then cordsys_param is given as a data set attribute, otherwise it is given as a global attribute.

for `geographical` (GDS octet 6: 0):
none

for `polar_stereographic` (GDS octet 6: 5):
0 North pole (0) / South pole (1) is on the projection plane (GDS octet 27)
1 orientation of the grid (GDS octets 18-20)
2, 3 Earth's radius in meters (GDS octet 17, bit 2)

for `space_view` (GDS octet 6: 90):
0 apparent diameter of earth in grid lengths in x-direction (GDS octets 18-20)
1 apparent diameter of earth in grid lengths in y-direction (GDS octets 21-23)
2 longitude of sub-satellite point (GDS octets 14-16)
3 latitude of sub-satellite point (GDS octets 11-13)
4 orientation of the grid (GDS octets 29-31)
5 altitude of the camera from the earth's centre in units of the earth's radius (GDS octets 32-34)
6, 7 Earth's radius in meters (GDS octet 17, bit 2)

for `shifted_pole` (GDS octet 6: 10):
0 longitude of the South pole of the rotated grid (GDS octets 36-38)
1 latitude of the South pole of the rotated grid (GDS octets 33-35)
2 angle of rotation (GDS octets 39-42)
3 longitude of pole of stretching (GDS octets 46-48)
4 latitude of pole of stretching (GDS octets 43-45)
5 stretching factor (GDS octets 49-52)
6, 7 Earth's radius in meters (GDS octet 17, bit 2)

The remaining parameters in the GDS are included as `step` and `start_value` as dimension attributes.

`unit` (implicitly defined by (PDS octet 9)) units used for the contents of the data.

`scale_factor`
`add_offset`
These parameters are used to scale the data to n-bits integers (n<=16). The original values are recalculated as: y = scale * (yi - offset).

`precision` Number of bits used to scale the data;

`n_scale_factor`
`n_add_offset`
`n_precision`
These parameters were introduced to combine asimof data sets into a single 3-dimensional data set. `n_scale_factor` and `n_add_offset` are arrays with size equal to the z-dimension. They are used to scale the data to n-bits integers (n<=16) similar to `scale_factor` and `add_offset`, except that for each horizontal plane specific scaling factors have been defined;

`ttype` (GDS octet 21) "forecast", "analysis_un", "analysis", "valid_range", "average", "accumulation", "difference", "valid_time", "climatological"
`analysis_un` stands for uninitialised analysis product or image product (P1 = 0);

`trange` (PDS octets 19-20) Time range.Time is expressed as a single number with the format: `yyyymmddhhhh`. `trange` is calculated from PDS octet 21. A point of time is indicated by `trange[0]=trange[1]`.

`zlevel` "isobaric", "msl", "ht_msl+", "ht_grd+", "sigma", "isentropic", "ocean", "atmosphere", "surface", "cld_base", "cld_top", "0_isotherm", "adiabat_condens", "max_wind", "tropopause", "nominal_top", "sea_bottom"
(PDS octet 10) In 3-D data sets, `zlevel` is replaced by dimension attribute `long name`;

`zrange`	vertical extent of the data. A plane is indicated by `zrange[0]=zrange[1]`. In 3-D data sets, `zrange` is replaced by dimension attribute `range` (z-dimension);
`zunit`	(PDS octets 11-12) units of `zrange`. In 3-D data sets, `zunit` is replaced by dimension attribute `unit` (z-dimension). All units are converted to SI units. Sigma levels are given as fractions. Model layers below ground level are indicated by 0, -1, -2 instead of 0, 999, 998 (hirlam definition);
`start_value`	(GDS octet 11-13; 14-16. for `shifted_pole`: GDS octet 24-25, 26-27) Latitude or longitude of the first grid point in the array;
`step`	grid size + orientation. If `step` is negative it implies that scanning mode is reversed to the direction of the projection coordinate system. In all cases the x-coordinate or y-coordinate of the last grid point equals `start_value + (N-1)*step`; for `geographical`: GDS octet 11-13, 14-16; for `polar_stereographic`: GDS octet 21-23, 24-26; for `space_view`: GDS octet 24-25, 26-27; for `shifted_pole`: GDS octet 28 bit 1-2 (±1).

The remaining GRIB header fields are translated as follows:

PDS octet 7:	always assumed 255 (GDS required);
PDS octet 8:	always assumed 64 (GDS given);
PDS octet 18:	implicitly defined by `trange`;
GDS octet 7-8:	implicitly given as the size of dimension 1 (use HDF function: `SDgetinfo()` to obtain this information);
GDS octet 9-10:	implicitly given as the size of dimension 2;
GDS octet 17:	(bit 5) if bit 5 = 0, then the data are reversed before saving (i.e. the directions are set always relative to the direction of x-axis and y-axis as in Hirlam);
GDS octet 28:	(bit 1-2) scanning mode given as the sign of dimension attribute `step`; (bit 3) if bit 3 = 1, then the data are transposed before saving (i.e. the data are always stored column major as in C).

Data Storage

The data are stored with an n-bits accuracy (n<=16) and compressed using the gzip encoding algorithm (default option of the HDF I/O library). If n<=8 then the data are stored physically as unsigned chars, otherwise the data are stored as unsigned shorts. The original data values can be recalculated as:

data_value = scale_factor *(array_value - add_offset)

The HDFg I/O routines will handle this conversion in the background.

Note:
The keywords `scale_factor` and `add_offset` do not affect the data storage. These attributes may always be included afterwards to modify the interpretation of the data. E.g. if the data are stored as bytes representing percentages, then including the scale factor `0.01` will cause the data to be interpreted as fractions.

Converting data from HDF to asimof

`hdf2asim` was provided to preprocess HDF files for Hirlam input. Its functionality is therefore limited to the basic needs. `hdf2asim` reads an HDF data set and copies its data to an asimof file. If this file does not exist then a new one is created, otherwise the data are appended. All GRIB header values, except for the fields 9 to 12, are set by default. The parameters describing the data set (fields 9-12) are user-supplied.

Usage:

```
hdf2asim hdffile 'ds1 param1 levtype1 level1 [ds2 ...]' asimof
```

`ds1, ds2,...`	HDF data set names
`param1, ...`	parameter classifying the data set (PDS octet 9)
`levtype1,...`	level type (PDS octet 10)
`level1,...`	level value (PDS octet 11-12)

The number of data sets that can be converted by a single call of `hdf2asim` is unlimited.
Example: `hdf2asim clim.hdf 'pressure 1 103 0' clim.asim`.

Tools to Convert Plain Data to HDF

The following three tools can be used to store the data in an HDF file:

```
cpset
addattr
addset
```

cpset

`cpset` is used to copy one data set from one HDF file to another. All attributes related to this data set are copied as well.

Usage:

```
cpset [-f][-dlayer][-ldataset {dimlist}][-ldim+name] infile dataset \
  outfile [dataset_out]
```

Options:

`-dlayer` copy only the layer indicated from a 3D input data set (first layer: `layer 0`);
`-l` option used to link or name the indicated:
`-ldataset {dimlist}`
link the dimensions indicated in `dimlist` to the dimensions of an existing data set;
`-ldim+name` rename the dimension to the name indicated. Both `-l` options are mutually exclusive;
`-f` If this option is set then the output is stored as floats;
`dataset_out` output data set name. If no output data set is given than the data set name used is copied from the input file.

Example:
`in.hdf` contains the data sets `soil_moist` (multi-layer) and `pressure` (single-layer). The first call to `cpset` copies the first layer of `soil_moist` to `out.hdf` as `soil_moist_0` and renames the dimensions to `longitude` and `latitude`. The second call copies `pressure` and links the dimensions to the dimensions of `soil_moist_0`. Consequently, the dimension names of `pressure` will also be `longitude` and `latitude`:

```
cpset -d0 -l1+longitude -l2+latitude in.hdf soil_moist out.hdf soil_moist_0
cpset -lsoil_moist_0 1 2 in.hdf pressure out.hdf
```

addset

`addset` adds a plain data set to an HDF file. The default format of the input data is binary. The data are read as a such without modifying its contents. Consequently, `numtype` should always represent the actual contents. Converting the data on a different machine than where they were written may lead to a wrongly interpreting the data.

ASCII data are read as a single stream of floats and converted to the data type indicated. Conversion takes place according to the standard rules. E.g. if the input file contains floats and `num_type` is set to `UINT8` then the values are truncated to an integer modulus 256. Note that there are no warning messages if values exceed the data range. Tabs, blank spaces and end-of-line are accepted as delimiters.

Usage:

```
addset [-t][-ASC][-a] hdf_file data_set infile numtype dim1..dimN
```

Options:

`-t`	this indicates that the data are stored top-down. This means that the first line of the data represents the top line of an image. Setting this option means the the image will be flipped vertically. This option takes effect for 2D data sets only;
`-ASC`	indicates that the input data are ASCII;
`-a`	indicates that the new data are appended to the output file. By default the HDF file is replaced;
`infile`	input file;
`numtype`	number type can be: `INT8`, `UINT8`, `INT16`, `INT32`, `FLOAT32` or `FLOAT64`;
`dim1..dimN`	are the dimension sizes of the input file.

addattr

`addattr` adds an attribute to an existing HDF file.

Usage:

```
addattr hdf_file [dataset dim] key numtype attr
```

`dataset` indicates the data set in `hdf_file` to which the attribute is included. Adding a data set attribute is indicated by setting `dim` to `0`, whereas `dim` greater than `0` refers to the dimension. A file attribute is indicated by omitting both `dataset` and `dim`.
`key` is the attribute name.
`numtype` can be `INT8`, `INT16`, `INT32`, `FLOAT32`, `FLOAT64` or `CHAR8`.
`attr` is the contents of the attribute and may contain an arbitrary list of elements or a string. If the attribute contains a string, then its length equals the length of the string plus 1. The last character contains the delimiter `'\0'`. `attr` must be placed between quotes if it consists of more than a single word.

Example:
The following statement adds keyword `grid_size` to `data_set`. `grid_size` is a single float equal to 0.5:

```
addattr in.hdf data_set 1 grid_size FLOAT32 0.5
```

The following statment adds some comments to a data set:

```
addattr in.hdf data_set 0 comment 'output created by model X'
```

Reference Guide to HDFg

General remarks

- The underlaying routines of the Fortran HDFg library are the C-library. Therefore, character strings must be closed with `char(0)`. Instead of, e.g., `"filename"` you should write `"filename"//char(0)`. Note that `"filename"` may not contain trailing blanks. If `"filename"` is stored in a string of 20 characters (`character*20 string`) then you should write: `string(1:8)//char(0)` instead. Also check how the compiler treats Fortran strings.
- Some HDFg functions accept `NULL`-pointers as arguments – unknown to Fortran – indicating that the output returned to this argument is not needed (e.g. as in `G_SDSinfo()`) or if the default value is used (as in `G_SDSw()`). Fortran programmers can use `null` instead. `null` is not a pointer, but a constant defined in `hdfg.inc` as `-32768` (`SHRT_MIN`).
- Functions needing dynamical memory allocation are defined for C only.
- Reading and riting the data from Fortran is handled differently from the standard NCSA library. Before storing or after recovering the data are they transposed. Consequently, a data set written by the HDFg library in Fortran and read by a C program look identical. This is not the case when using the original HDF library. In this case the data are transposed compared to the original data in the sense that a Fortran array `A(d1,d2)` will be interpreted as `A[d2][d1]`. Using the HDFg library overcomes this problem by transposing the data first.

Naming Conventions

`I` indicates an input argument.
`O` indicates an output argument.
`IO` indicates that this variable must be initialised first. Its contents is modified by the function.
`<any>` indicates that this (pointer) variable can be of any type. This corresponds to `void` in C.

Function Definitions

libHDFg contains functions to simplify I/O to and from HDF files containing scientific data sets. The following functions are available:

C	Fortran	Description
Opening/Closing files		
`G_SDcreate()`	`G_SDCREATE()`	creates an HDF-file for writing
`G_SDopen()`	`G_SDOPEN()`	opens an HDF-file for reading and writing. A write-protected file is opened as read-only
`G_SDclose()`	`G_SDCLOSE()`	closes an HDF-file and updates the file.
Reading/Writing SDS		
`G_SDSr()`	`G_SDS_R()`	reads the contents of an SDS.
`G_SDSw()`	`G_SDS_W()`	writes the contents of an SDS. If the data set already exists, then the data are updated.
`G_SDSa()`	`G_SDS_A()`	appends a data block to an SDS with unlimited dimension.
`G_SDSrdgrib()`	`n.a.`	simplifies reading an SDS data set derived from GRIB data.
Query functions		
`G_SDSinfo()`	`G_SDSINFO()`	returns rank, dimensions, number type and sizes of an SDS
`G_SDSname()`	`G_SDSNAME()`	reads the next SDS name in an HDF file.
`G_SDScount()`	`G_SDSCOUNT()`	returns the number of scientific data sets in the file.
`G_SDisunlimited()`	`G_SD_ISUNLIMITED()`	checks whether an SDS contains an unlimited dimension.
`Reading/Writing attributes`		
`G_AttR1()`	`G_ATT_R1()`	reads a single attribute from an HDF file.
`G_AttsR()`	`n.a.`	reads a linked list of attribute from an HDF file.
`G_AttsFR()`	`n.a.`	initialises a list of attributes from a resource file.
`G_AttW1()`	`G_ATT_W1()`	writes a single attributes to an HDF file.
`G_AttsW()`	`n.a.`	writes a linked list of attribute to an HDF file.
`G_AttsFW()`	`G_ATTS_FW()`	appends the attributes described in a resource file, to an HDF file.
Dimension-related functions		
`G_SDSdimpurge()`	`n.a.`	removes all dimensions of size 1 in an SDS structure.
`G_SDSdim1()`	`n.a.`	adds dimensions of size 1 to an SDS structure.
`G_DIMsetname()`	`G_DIM_SETNAME()`	renames the dimension name.
`G_DIMlink()`	`G_DIM_LINK()`	combines the dimension meta information of two data sets.
`G_DIMsetscale()`	`G_DIM_SETSCALE()`	writes the dimension scale indicated.
`G_DIMgetscale()`	`G_DIM_GETSCALE()`	reads the dimension scale indicated.
Utility functions		
`attlist_append()`	`n.a.`	allocates memory in a linked list of attributes for a new attribute to be appended.
`attlist_free()`	`n.a.`	frees the memory occupied by a linked list of attributes.
`attlist_getlink()`	`n.a.`	returns a pointer to a link in a list containing the keyword
`cij()`	`n.a.`	returns the position of an element in an array corresponding to the coordinates indicated.

G_SDcreate

C
```
int32 G_SDcreate (char *fname)
```

Fortran
```
integer function G_SDCREATE (fname)
I  character*(*)  fname
```

`G_SDcreate()` creates a new HDF-file and returns a file ID (from now on referred to as `sd_id`). An existing HDF-file with the same name is overwritten. The fill mode is set to `SD_NOFILL` (`SDsetfillmode()`).

Function output
`sd_id` or `FAIL`

G_SDopen

C
```
int32 G_SDopen (char *fname)
```

Fortran
```
integer function G_SDOPEN (fname)
I  character*(*)  fname
```

`G_SDopen()` opens an existing HDF-file for reading and writing and returns a file ID. If the file is write-protected, then the file is opened as read-only.

Function output
`sd_id`, `NO_HDF` or `FAIL`

G_SDclose

C
```
void G_SDclose (int32 sd_id)
```

Fortran
```
procedure G_SDCLOSE (sd_id)
I  integer*4     sd_id
```

`G_SDclose()` closes an open HDF-file and saves the changes made during the session. If an HDF-file is not explicitly closed, then all changes made during the session are lost.

G_SDSw

C

```
int G_SDSw (int32 sd_id, char *sds_name, SDS *sds, lyr_info *info,
            void *fillvalue, int np, ...)
```

Fortran

```
integer function G_SDS_W (sd_id, sds_name, numtype, data, rank, arr_size,
sds_size, offset, stride, fillvalue, np, ...)
```

I	integer*4	sd_id	file ID
I	character*(*)	sds_name	data set name
I	integer*4	numtype	data set number type
I	<any>	data()	data array
I	integer*4	rank	number of dimensions of data
I	integer*4	arr_size()	dimension sizes of data
I	integer*4	sds_size()	dimension sizes of the data set or subset to be created, NULL or {SD_UNLIMITED}
I	integer*4	offset()	first element in the data set to be written or NULL
I	integer*4	stride()	number of lines to be skipped per dimension or NULL
I	<any>	fillvalue	value to indicate no-value data or NULL
I	integer	np	size of precision, range_max and range_min
variable list arguments:			
I	char()	precision	number of bits to trunk the data values (<=16) (array)
I	<valid type>	range_max	upper boundary of scale (array)
I	<valid type>	range_min	lower boundary of scale (array)

G_SDSw() writes data to data set sds_name in file sd_id. If the data set already exists then the data are overwritten, otherwise a new data set is created. lyr_info is a structure containing the following fields:

```
typedef struct {
  int32 *size;
  int32 *offset;
  int32 *stride;
  int32 *chunks;
} lyr_info;
```

The first three fields correspond to sds_size, offset, and stride in the Fortran call. chunks is an array to define the chunk sizes to split up the array. Any of the fields can be set to NULL, if they are not required. If none of the information in info is needed, then info can be set to NULL. If info.offset or offset is set to NULL, it is assumed to be {0,0,..0}. If info.stride or stride is set to NULL, it is assumed to be {1,1,..1}. The dimensions of the data set to be created are given by sds_size. These dimensions can be different from the dimensions of the array (sds->dim (C) or arr_size in Fortran), e.g. if you want to write a layer in a 3-dimensional array. If they do agree or if you want to (over)write (parts of) an existing SDS, then sds_size or info.size can be set to NULL. If you want to create a data set with unlimited dimension, then sds_size can be set to {SD_UNLIMITED}. In this case, the data set to be written is assumed to be the first data block.
If the rank of data is less then the rank of the SDS, then the missing dimension must be indicated by 1. For example, a horizontal plane in a 3D data set is denoted by {10,20,1} instead of {10,20} and rank equals 3. offset and stride must have the same rank as sds_size.

If the data set contains invalid data, then they must be indicated by fillvalue. fillvalue must be of the same type as the data set. If fillvalue is included, then the attribute _FillValue is set.

np is the array size of precision. np can have the following values:

0: the original (unscaled) data are stored. In this case the variable list arguments are obsolete and can be deleted;

1: The original data are scaled before they are stored. They can be recalculated as:
```
scale_factor *(stored_data - add_offset);
```
size dimension 3: The original data are scaled. Each horizontal layer is scaled using a distinct precision. Layer n of the original data can be recalculated as:
```
n_scale_factor[n] *(stored_data - n_add_offset[n]);
```

precision is the number of bits used to trunk the data values. E.g. a 10-bits precision corresponds to 3 significant digits. If you define precision for data sets of type int or char, then the data values are truncated to the number of bits indicated. Scaling is applied to data of type float or double only (DFNT_FLOAT32 or DFNT_FLOAT64). If np is set to a value larger than 0, then {n_}add_offset, {n_}scale_factor and {n_}scale_factor_err will be included to the file as data-set attributes. {n_}scale_factor_err equals the maximum potential error as a result of scaling the data. The data are stored in the smallest possible data type (8 or 16 bits unsigned integer).
The data-set values are scaled between the maximum and minimum value of the data set. However, if the data set is appendable (i.e. contains an unlimited dimension), then scale_factor and add_offset cannot be determined in advance, because new data blocks could be appended not fitting the range. Therefore, the range for scaling the data can also be included explicitly as range_max and range_min. This range is user-defined and must be choosen as small as possible (to obtain the smallest error), but large enough to contain all values that will be appended. The range must be given in the same number type as the data set itself.
If precision is set to 1, then a bitmap is created (variable bit-length is set to 1). The bitmap corresponds to the first bit, representing integer value 1.

If a fill value has been defined, then it will no longer match the scaled data. G_SDSw() will re-define fill_value as 0. Hence, the fill value in the restored data set equals -add_offset*scale_factor.

The C version and Fortran version of the function are slightly different although the results are the same. In Fortran the record fields of SDS must be given explicitly as numtype, data(), rank and arr_size.

Warnings

- Do not use SHRT_MIN (null) as fillvalue (reserved for Fortran null).
- Always set sds_size to NULL if you write to an existing data set.

Examples

In the following examples Fortran programmers can read arr_size for sds.dim, and rank for sds.rank. There is no difference in the initialisation.

Create a new data set of dimensions {2,3,4} and write its contents:
Initialise (using pseudo code):
```
sds.rank = 3
sds.dim  = {2,3,4}
```
Call:
```
result = G_SDSw (sd_id, name, &sds, NULL, NULL, 0);
```

Similarly in Fortran:
```
      include "hdf.inc"
      integer*4 ierr
      integer*4 numtype
      real      data(2,3,4)
      integer*4 dim(3)
      data dim /2,3,4/
..
      ierr = G_SDS_W (sd_id, name, DFNT_FLOAT32, data, 3, dim,
     * NULL, NULL, NULL, NULL, 0)
```

Create a new data set and write the bottom layer:
Initialise:

```
sds.rank = 3
sds.dim  = {2,3,1}
info.size = {2,3,4}
```
(Fortran: sds_size)

The remaining fields of `info` must be set to `NULL`

Call:

```
result = G_SDSw (sd_id, name, &sds, &info, NULL, 0);
```

Similarly in Fortran:

```
      include "hdf.inc"
      integer*4 ierr
      integer*4 numtype
      real      data(2,3,1)
      integer*4 dim(3), sdsdim(3)
      data dim    /2,3,1/
      data sdsdim /2,3,4/
..
      ierr = G_SDS_W (sd_id, name, DFNT_FLOAT32, data, 3, dim,
     * sdsdim, NULL, NULL, NULL, 0)
```

Write the second layer to the same data set:
Initialise:

```
sds.rank = 3
sds.dim  = {2,3,1}
info.offset = {0,0,1}
```
(Fortran: offset)

The remaining fields of `info` must be set to `NULL`

Call:

```
result = G_SDSw (sd_id, name, &sds, &info, NULL, 0);
```

Write the first data block to a time series of data block:
Initialise:

```
sds.rank = 3
sds.dim  = {2,3,4}
info.size = {SD_UNLIMITED}
```
(Fortran: sds_size)

The remaining fields of `info` must be set to `NULL`

Call:

```
result = G_SDSw (sd_id, name, &sds, &info, NULL, 0);
```

Rewrite the second data block of a time series of data block:
Initialise:

```
sds.rank = 3
sds.dim  = {2,3,4}
info.offset = {1,0,0,0} /* the first dimension represents the time dimension */
```

The remaining fields of `info` must be set to `NULL`

Call:

```
result = G_SDSw (sd_id, name, &sds, &info, NULL, 0);
```

Rewrite the third layer in the second data block of a time series of data blocks:
Initialise:

```
sds.rank = 3
sds.dim  = {2,3,1}
info.offset   = {1,0,0,2}
```

The remaining fields of `info` must be set to `NULL`

Call:

```
result = G_SDSw (sd_id, name, &sds, &info, NULL, 0);
```

Write a data set in 10-bits precision.:
Initialise:
```
  precision = 10
```
Call:
```
  result = G_SDSw (sd_id, name, &sds, NULL, NULL, 1, &precision, NULL, NULL);
```

Similarly in Fortran:
```
      include "hdf.inc"
      integer*4 ierr
      integer*4 numtype
      real      data(2,3,4)
      integer*4 dim(3)
      data dim /2,3,4/
      character prec
..
      ierr = G_SDS_W (sd_id, name, DFNT_FLOAT32, data, 3, dim,
     * NULL, NULL, NULL, NULL, 1, 10)
```

Note that a variable list of arguments is not official Fortran, but can be applied because the underlaying language is C.

Function output
`FAIL` or `SUCCEED`

G_SDSa

C
```
int G_SDSa (int32 sd_id, char *sds_name, SDS *sds)
```

Fortran
```
integer function G_SDS_A (sd_id, sds_name, data, precision)
```

I	integer*4	sd_id	file ID
I	character*(*)	sds_name	data set name
I	<any>	data()	data array

`G_SDSa()` appends a data block to an existing data set. The data block must have identical dimensions and must be of the same data type as the first one. This is implicitly assumed. The fist data block of an appendable data set is created by using `G_SDSw()`.

Function output
`FAIL` or `SUCCEED`

G_SDSr

C
```
int G_SDSr (int32 sd_id, char *sds_name, SDS *sds, int32 numtype,
            lyr_info *info, int timestep, void &fillvalue)
```

Fortran
```
integer function G_SDS_R (sd_id, sds_name, data, numtype, size, offset, stride,
timestep, fillvalue)
```

I	integer*4	sd_id	file ID
I	character*(*)	sds_name	data set name
IO	integer*4	numtype	data set number type or 0
O	<any>	data	data array
IO	lyr_info	info	(C only)
I	integer*4	size	dimension sizes of the array to be read

I	integer*4	offset	first element in the data set to be read or NULL
I	integer*4	stride	number of lines to be skipped per dimension or NULL
I	integer*4	timestep	time step for which the data are read (1,2,...).
O	<any>	fillvalue	value indicating no-value data or NULL

G_SDSr() reads a data set and returns the array. If numtype = 0, then the data are returned in the original number type and the data-set number-type is returned (Fortran only). In C the number type is returned as part of sds. If the data are stored as scaled data then they will not be converted to their original values. If numtype differs from the actual data number-type, then the output is converted as indicated. This method is applicable to conversions between (unsigned) char, (unsigned) short, (unsigned) int32, float and double. If the number type indicated differs from the type in which the data are stored, then fillvalue is converted to the appropriate type. If the data set does not contain a fill value (keyword _FillValue), then fillvalue remains unchanged. It is assumed that _FillValue is of the same type as the data stored. If the data are scaled, then the no-data value must be 0.
G_SDSr() cannot be used to read multiple time steps at once.

The C version and Fortran version of the function are slightly different.

C:
The data set is returned as a variable of type SDS. If the entire data set must be read, then an empty SDS variable can be passed and info can be set to NULL. G_SDSr() will set its entire contents and will allocate memory to the fields. **If the SDS variable is recycled, be aware to free all memory first and set the fields sds.data and sds.dim to NULL before passing it to G_SDSr() as an argument.** If you want to read a subset of the array, then you must first set the dimensions of the subset to be read. In this case, memory is allocated only to sds.data. If info.offset = NULL then, by default, offset is set to {0,0,...}. If info.stride = NULL then, by default, stride = {1,1,...} (i.e. do not skip lines).
Note that the rank of the data set must be the same as the rank of the SDS, even though it represents a subset of lower dimensions. A horizontal plane in a 3D data set must be denoted e.g. by {10,20,1} instead of {10,20}. You can use the function G_SDSdim1() to include the extra dimensions and call G_SDSdimpurge() to remove them again. offset and stride must have the same rank as the SDS. If numtype is set, then sds.type is set to numtype, and the data values are converted to the type requested.
If info is not equal to NULL and all fields in info are set to NULL, then G_SDSr() returns the chunk sizes (warning: in this case memory must be allocated to info.chunks).

Fortran:
If you want to read the entire array you can set size to null. Be aware that there is enough memory allocated to data to contain the entire array. Subsets can be read by setting size, offset and stride different from null = {1,1,...} (i.e. do not skip lines). If you need information about the data set first, you can call G_SDSINFO(). Note that the rank of the data set must be the same as the rank of the SDS, even though it represents a subset of lower dimensions. A horizontal plane in a 3D data set must be denoted e.g. by {10,20,1} instead of {10,20}. offset and stride must have the same rank as the SDS.

Examples

Initialise:
```
SDS sds = {0, NULL, 0, NULL};
```
Call:
```
G_SDSr (sd_id, name, &sds, 0, NULL, 1, NULL); /* read the entire data set */
```

Read the top layer of the second time block. Data set size: {3,4,5}
Initialise:
```
sds.rank = 3
sds.dim  = {3,4,1}
info.offset = {0,0,4}
```

```
  info.stride = NULL
```
Call:
```
  G_SDSr (sd_id, sds_name, &sds, 0, &info, 2, NULL);
```

The following call will always convert the data to an array of floats, whatever the type of data stored:

```
  G_SDSr (sd_id, sds_name, &sds, DFNT_FLOAT32, NULL, 1, NULL);
```

Warning
Fortan programmers must be aware that the output is always given as a contiguous array. If you have defined

```
  real data(10,20)
```

and used it as the argument for `G_SDS_R()`to read an array of size`(5,10)`, then you cannot access the elements properly.

C programmers can use the function `cij()`:

```
  sds.value[cij(sds.rank,sds.dim,i,j)]
```

to get access to element `[i][j]` in the array.

`cij()` is part of libHDFg.a.

Function output
`FAIL` or `SUCCEED`

G_SDSrdgrib

C
```
int G_SDSrdgrib (int32 sd_id, int id, char *ext, int layer, lyr_info *info,
              SDS *sds, int mode)
```

Fortran
```
integer function G_SDSrdgrib (sd_id, id, ext, layer, data, dim mode)
```

I	integer*4	sd_id	file ID
IO	integer*4	numtype	data set number type or 0
I	character*(*)	ext	extra extension appended to the data set name (see Naming Conventions)
I	integer*4	layer	layer to be read
I	lyr_info	info	(C only; optional) or NULL
O	SDS	sds	output data (C)
O	real	data(*)	data array (Fortran)
O	real	dim(*)	dimensions of the array (Fortran)
I	integer*4	mode	0 if an FAIL should be returned after a reading error has. occurred. 1 to stop the application (return value: exit(1))

`G_SDSrdgrib()` reads a data set from a file created by `asim2hdf` or a file using the same naming conventions for the dats sets. The id and extension must be according to the naming conventions described above and in the appendix. The data returned are always of type `floats`.
If you do not want to set `info` then you can insert `NULL`. If the data set contains several layers then, if `layer` is set to `-1`, the entire array is returned. If you want a specific layer only, then the layer to be read can be indicated by setting `layer` to a legal value (`0` <= `layer` < # layers). `info` can be set independent of `layer`. If `info` is set, then if will overrule `layer` if its contents contradicts `layer`, e.g. if the field

info.size does not correspond to a single horizontal layer. If the data set is 2-dimensional, then layer is disregarded. See G_SDSw() for more details on lyr_info.
If mode is set to 1 or CONT_ON_ERR, then G_SDSrdgrib() returns error code FAIL or SUCCEED. If mode is set to 1 or EXIT_ON_ERR then the application will stop with an error message.

Notes
Note that the contents of sds must be reset in the same way as when applying G_SDSr().
If you want to use G_SDSrdgrib() then you must include grib_def.h in addition to HDFg.h.

Examples

```
#include "gribdef.h"
...
G_SDSrdgrib (sd_id, PRESSURE, "", 0, NULL, &sds, EXIT_ON_ERR);

if (G_SDSrdgrib (sd_id, WIND_V, "_=", 31, NULL, &sds, CONT_ON_ERR) == FAIL) {
  do your own error handling...
}
```

The first call will return the contents of data set "pressure" and exit if it cannot be read. The second statement will return the 32rd layer of the data "wind_v_=", which contains the model surface layer v-component of the wind.

G_AttW1

C

```
int G_AttW1 (int32 sd_id, char *sds_name, int dim, char *keyword,
             int32 numtype, int32 count, void *attr);
```

Fortran

```
integer function G_ATT_W1 (sd_id, sds_name, dim, keyword, numtype, count, attr)
```

I	integer*4	sd_id	file ID
I	character*(*)	sds_name	data set name or ""
I	integer	dim	dimension or 0
I	character*(*)	keyword	keyword
I	integer*4	numtype	attribute number type
I	integer*4	count	attribute length
I	<any>	attr	attribute contents

G_AttW1() writes a single attribute to an HDF file. sd_id is obtained from G_SDopen() or G_SDcreate(). Existing attributes with the same name are overwritten. The level to which the attribute is appended, is indicated by combinations of sds_name, and dim as follows:

file	sds_name equals "" (C) or char(0) (Fortran). dim is disregarded;
data set	sds_name indicates a data set name and dim = 0;
dimension	sds_name indicates a data set name and dim > 0;

The attribute is defined by a keyword, number type (see Table 1), size, and contents.

Function output
FAIL or SUCCEED

G_AttsW

C
```
int G_AttsW (int32 sd_id, char *sds_name, int dim, attribute *attr)
I  sd_id
I  *sds_name
I  dim
I  *attr
```

`G_AttsW()` writes a list of attributes to an HDF file. The level is defined by `sds_name` and `dim` as indicated above. Invalid links (`len = 0` or `keyword = '\0'`) are skipped. Existing attributes with the same name are overwritten. The level to which the attribute is appended, is indicated by combinations of `sds_name`, and `dim`, as described above.

Function output
`FAIL` or `SUCCEED`

G_AttsFW

C
```
int G_AttsFW (int32 sd_id, char *sds_name, int dim, char *fname)
```

Fortran
```
integer function G_ATTS_FW (sd_id, sds_name, dim, fname)
I  integer*4       sd_id        file ID
I  character*(*)   sds_name     data set name or ""
I  integer         dim          dimension or 0
I  character*(*)   fname        file name containing the attributes
```

`G_AttsFW()` reads the attributes from file and attaches the data to an HDF file. The level to which the attribute is appended, is indicated by combinations of `sds_name`, and `dim` as indicated above. Existing attributes with the same name are overwritten.

The resource file must have the following format:
- Each line contains the data of a single attribute.
- A line may not exceed 1023 characters.
- A keyword must be a single word not exceeding 31 characters.

Format:
```
  keyword type values
```
where:
`type` are the definitions given in Table 1 except for the `DFNT_`-prefix.
If `type = CHAR8` then the rest of the line is interpreted as the attribute contents.

Examples
```
date    INT32  1997 6 17
comment CHAR8 This is an HDF file
```

The first attribute will be initialised as an integer array of length 3: `{1997, 6, 17}`, the second one as the character string: `"This is an HDF file"`.

Function output
`FAIL` or `SUCCEED`

G_AttR1

C
```
int G_AttR1 (int32 sd_id, char *sds_name, int dim, char *keyword,
             int32 *numtype, int32 *count, void *attr)
```

Fortran
```
integer function G_ATT_R1 (sd_id, sds_name, dim, keyword, numtype, count, attr)
```

I	integer*4	sd_id	file ID
I	character*(*)	sds_name	data set name or ""
I	integer	dim	dimension or 0
I	character*(*)	keyword	keyword
O	integer*4	numtype	attribute number type or NULL
O	integer*4	count	attribute length or NULL
O	<any>	attr	attribute contents or NULL

`G_AttR1()` reads the contents of a single attribute named by `keyword`. The level is defined by `sds_name`, and `dim` as described above. `G_AttR1()` does not allocate memory to `attr`. If the information for `numtype`, `count`, or `attr` is not needed, you can replace them by `NULL`. To check the memory required to store the attribute you can set `attr` to `NULL`. The memory required is returned as the function value.

Example

```
size = G_AttR1 (sd_id, sds_name, dim, keyword, NULL, NULL, NULL);
if (size != FAIL) {
  attr = malloc (size);
  G_AttR1 (sd_id, sds_name, dim, keyword, &numtype, &count, attr);
}
```

Function output
attribute size (bytes) or `FAIL`

G_AttsR

C
```
int G_AttsR (int32 sd_id, char *sds_name, int dim, attribute *attr)
```

I	int32	sd_id	file ID
I	char	*sds_name	data set name or ""
I	int	dim	dimension or 0
IO	attribute	*attr	linked list of attributes

`G_AttsR()` reads the attributes at a level defined by `sds_name` and `dim` as described above. If `attr` is empty `{"", 0, 0, NULL, NULL}`, then all the attributes at that level are returned as a linked list. Memory is allocated dynamically. If `attr` is not empty, then only the attributes that are listed are read. If keywords in the list do not match, then the memory allocated to the field `value` is released and the field `count` is set to `0`.

Function output
`FAIL` or `SUCCEED`

G_AttsFR

C
```
int G_AttsFR (char *fname, attribute *attr)
```

I	char	*fname	file name

`IO attribute *attr`	linked list of attributes

`G_AttsFR()` reads the attributes stored in a resource file as described above (`G_AttsFW`). `G_AttsFR()` returns them in a linked list. `G_AttsFR()` can be used to create templates.
`attr` must be an empty list: `{"", 0, 0, NULL, NULL}`.

Function output
`FAIL` or `SUCCEED`

G_SDisunlimited

C
```
int G_SDisunlimited (int32 sd_id, char *sds_name)
```

Fortran
```
integer function G_SD_ISUNLIMITED (sd_id, sds_name)
```

I	`integer*4`	`sd_id`	file ID
I	`character*(*)`	`sds_name`	data set name

`G_SDisunlimited()` returns `1` if the first dimension of the array is appendable and `0` if it is not.

Function output
`FAIL`, `0` (false) or `1` (true).

G_SDSdimpurge

C
```
void G_SDSdimpurge (SDS *sds)
```

`G_SDSdimpurge()` removes the dimensions of size 1 in the `SDS` field `dim`, and adjusts the rank. The memory allocated to `sds.dim` remains unchanged.

Example

```
sds.dim = {10,1,30}
```
then
`G_SDSdimpurge(&sds)` results in:
`sds.dim = {10,30}` and `sds.rank = 2`

G_SDSdim1

C
```
void G_SDSdim1 (SDS *sds, int dim)
```

`G_SDSdim1()` inserts value 1 for dimension `dim` and increases the rank by 1. The memory occupied by `sds.dim` is re-allocated and increased by 1. This function can be used to temporarily match the dimensions of `sds` with the dimensions of a scientific data set for I/O.

Example

```
sds.dim = {10,30}
```
then
```
G_SDSdim1(&sds,2);
G_SDSdim1(&sds,1);
```

results in:
`sds.dim = {1,10,1,30}` and `sds.rank = 4`

G_DIMsetname

C
```
int32 G_DIMsetname (int32 sd_id, char *sds_name, int dim, char *nw_dimname)
```

Fortran
```
integer function G_DIM_LINK (sd_id, sds_name, dim, nw_dimname)
```

I	integer*4	sd_id	file ID
I	character*(*)	sds_name	data set name
I	integer	dim	dimension
I	character*(*)	nw_dimname	dimension name

`G_DIMsetname()` is used to give the dimensions to an appropriate name. The default dimension names are `fakeDim1` etc. Be aware that the dimensions must be renamed first before any dimension attribute is written.

Function output
`FAIL` or `SUCCEED`

G_DIMlink

C
```
int32 G_DIMlink (int32 sd_id, char *sds_name, char *sds_link_to, int dim)
```

Fortran
```
integer function G_DIM_LINK (sd_id, sds_name, sds_link_to, dim)
```

I	integer*4	sd_id	file ID
I	character*(*)	sds_name	data set name
I	character*(*)	sds_link_to	data set name
I	integer	dim	dimension

`G_DIMlink()` links the dimension of data set `sds_name` to the same dimension of data set `sds_link_to`. As a result, both data sets will point to the same block of meta-information for dimension `dim`. Changes in the meta-information for dimension `dim` of either data sets will affect both. Linking dimensions reduces the file size. Be aware that the dimensions must be linked before any dimension attribute is written. Once the dimensions are linked, they cannot be unlinked any more. Use `G_DIMsetname()` to change the dimension names.

Function output
`FAIL` or `SUCCEED`

G_DIMsetscale

C
```
int32 G_DIMsetscale (int32 sd_id, char *sds_name, int dim, void *scale, int32
type)
```

Fortran
```
integer function G_DIM_SETSCALE (sd_id, sds_name, sds_name, dim, scale, type)
```

I	integer*4	sd_id	file ID

I	character*(*)	sds name	data set name
I	integer	dim	dimension
I	float	scale()	scale values
I	integer*4	type	data set type of scale

G_DIMsetscale() writes the scale to the dimension of the data set indicated. scale is defined as an array of floats of size equaling the dimension size. If you want to include a scale of a different number type then you must use the NCSA HDF functions.

Function output
FAIL or SUCCEED

G_DIMgetscale

C
```
int32 G_DIMgetscale (int32 sd_id, char *sds_name, int dim, void **scale,
                     int32 *type)
```

Fortran
```
integer function G_DIM_GETSCALE (sd_id, sds_name, sds_name, dim, scale)
```

I	integer*4	sd_id	file ID
I	character*(*)	sds_name	data set name
I	integer	dim	dimension
O	float	scale()	scale values
IO	integer*4	type	data set type of scale

G_DIMgetscale() reads the scale of the dimension indicated and returns the values to variable scale. In C memory is allocated to scale. In Fortran, it is assumed that scale is large enough to store all the values. If type is set to 0, then the scale is returned in the data type as stored, and its number type is returned as type. If type has a value as defined in Table 1, then scale is converted to this data type.

Function output
FAIL or SUCCEED

G_SDSinfo

C
```
int G_SDSinfo (int32 sd_id, char *sds_name, int32 *rank, int32 *dims,
               int32 *numtype)
```

Fortran
```
integer function G_SDSINFO (sd_id, sds_name, rank, dims, numtype)
```

I	integer*4	sd_id	file ID
I	character*(*)	sds_name	data set name or ""
O	integer*4	rank	number of dimensions or NULL
O	integer*4	dims	dimension size or NULL
O	integer*4	numtype	data set number type or NULL

G_SDSinfo() returns the rank, dimensions, number type and the data set size including the time dimension. This function does not indicate whether the first dimension is of unlimited length. You can use the function G_SDisunlimited() to figure this out. If you do not want to query either rank, dims, or numtype, you can replace them by NULL. The function returns the size needed to store the entire array.

Function output
data set size (bytes) or FAIL

G_SDSname

C
```
int G_SDSname (int32 sd_id, int32 *ref, char *sds_name)
```

Fortran
```
integer function G_SDSNAME (sd_id, ref, sds_name)
```

I	integer*4	sd_id	file ID
IO	integer*4	ref	reference number for the data-set name to be read
O	character*(*)	sds_name	data set name

G_SDSname() returns the data set names stored in file sd_id. ref is a variable for internal use. The first name is obtained by setting ref to 0.

Example
The following code shows how to get a list of all the data sets stored in an HDF file.

```
ref = 0;
while (G_SDSname(sd_id, &ref, name) != FAIL) {
  printf ("%s\n", name);
}
```

Function output
FAIL or SUCCEED

G_SDScount

C
```
int G_SDScount (int32 sd_id)
```

Fortran
```
integer function G_SDSCOUNT (sd_id)
```

I	integer*4	sd_id	file ID

G_SDScount() returns the number of data sets stored in HDF-file sd_id.

Function output
number of data sets or FAIL

cij

C
```
int32 cij (int rank, int32 *dim, ...)
```

I	rank	number of dimensions
I	*dim	dimension sizes
I	...	array coordinates

cij() returns the position of an element in an array. The variable list contains the coordinates of the element in an array of dimensions dim.

Example
```
sds.rank = 2
sds.type = DFTN_FLOAT32
sds.dim = {10,5}
```
then

`sds.data[cij(sds.rank, sds.dim, 2, 3)]` corresponds to `data[2][3]`, if `sds.data` were typecast to an array of size `{10,5}`.

attlist_append

C
```
void attlist_append (attribute **attr, char *keyword, int32 count,
                     int32 numtype, void *values, attribute *found)
```

IO attr	reference to a pointer to a linked list
I keyword	keyword of the attribute to be appended
I count	attribute length
I numtype	attribute number type
I values	pointer to the attribute contents to be written or `NULL`
O found	pointer to the attribute appended or `NULL` (if not required)

`attlist_append()` appends a new attribute to a list and allocates memory or finds the link containing `keyword`. The keyword, number type and length are initialised. If the list already contains an attribute `keyword`, then this link is re-initialised (`count` can be equal to `0`). `attlist_append()` returns a pointer to the initialised or appended attribute. If the list is empty, then the first link is initialised. If `values = NULL`, then `attlist_append()` creates an empty link and allocates memory for field `value`. `found` points to the link that has been initialised, hence `found->keyword = keyword`.

attlist_getlink

C
```
void attlist_getlink (attribute **attr, char *keyword)
```

IO attr	reference to a pointer to a linked list
I keyword	keyword of the attribute to be searched

`attlist_getlink()` returns the link in the list containing `keyword`. If the keyword is not found then `NULL` is returned.

Example

`attlist` is a linked list of attributes containing `valid_range`:

```
attribute *range;
range = attlist;  /* initialisation */
attlist_getlink (&range, "valid_range");
```

will result in:

`range->keyword = "valid_range"` etc.

attlist_free

C
```
void attlist_free (attribute *attr)
```

`attlist_free()` releases all memory occupied by a linked list of attributes.

Appendix 1

Data set names used for the GRIB parameter code (version: PDS field 9 = 1, i.e. for Hirlam). Codes 1-127 are universal codes. The same names in capital can be used as identifiers in the `grib_param` list, e.g.: `grib_param[PRESSURE]` equals `grib_param[0]` or `"pressure"`.

1	pressure	51	hum_specific	103	waves_wind_prd
2	press_msl	52	hum_relative	104	waves_swell_dir
3	press_tendency	53	hum_mix_ratio	105	waves_swell_sign_ht
6	geopotential	54	prec_water	106	waves_swell_ht
7	geopotential_ht	55	vapor_press	107	1st_wave_dir
8	geometric_ht	56	sat_deficit	108	1st_wave_prd
9	sd_height	57	evaporation	109	2nd_wave_dir
11	temperature	59	prec_rate	110	2nd_wave_prd
12	T_virtual	60	thunder_prob	111	net_rad_sh_surf
13	T_potential	61	prec_total	112	net_rad_l_surf
14	T_pseudo_adiabatic_pot	62	prec_large_scale	113	net_rad_sh_toa
15	T_max	63	prec_convec	114	net_rad_l_toa
16	T_min	64	snowfall_rate	115	rad_long
17	T_dewpoint	65	snow_depth_acc	116	rad_short
18	dewpoint_depr	66	snow_depth	117	rad_global
19	lapse_rate	67	pbl	121	heat_flux_E
20	visibility	68	trans_thermocl_depth	122	heat_flux_H
21	radar_spectra_1	69	main_thermocl_depth	123	bound_lyr_dissip
22	radar_spectra_2	70	main_thermocl_anomally	124	momentum_flx_u
23	radar_spectra_3	71	cld_cov_total	125	momentum_flx_v
25	anomally_temp	72	cld_cov_convec	127	image_data
26	anomally_press	73	cld_cov_low	128	momentum_flx
27	anomally_geopot_ht	74	cld_cov_medium	170	forest_clearing
28	wave_spectra_1	75	cld_cov_high	171	forest_needle
29	wave_spectra_2	76	cld_water	172	forest_needle_sparse
30	wave_spectra_3	81	landmask	173	forest_loaf
31	wind_dir	82	sea_level_dev	174	forest_loaf_sparse
32	wind_speed	83	surf_rough	175	forest_mixed
33	wind_u	84	albedo	176	forest_bushland
34	wind_v	85	soil_temp	179	forest_undef
35	stream_func	86	soil_moist	180	agric
36	vel_potential	87	vegetation	181	bare_mountain
38	vert_vel_s_coord	88	salinity	182	barren
39	vert_vel_press	89	density	183	wetland_wet
40	vert_vel_geom	91	ice	184	wetland_dry
41	abs_vort	92	ice_thick	185	snow
42	abs_div	93	ice_drift_dir	186	agric_irrig
43	rel_vort	94	ice_drift_spreed	187	grassland
44	rel_div	95	ice_drift_u	188	urban
45	vert_shear_u	96	ice_drift_v	189	open_land_undef
46	vert_shear_v	97	ice_growth	195	soil_type
47	cur_wind_dir	98	ice_div	196	lakes
48	cur_wind_speed	100	waves_wind_swell_ht	197	forest
49	cur_wind_u	101	waves_wind_dir	198	open_land
50	cur_wind_v	102	waves_wind_sign_ht		

KNMI-PUBLICATIES, VERSCHENEN SEDERT 1996

Een overzicht van eerder verschenen publicaties, wordt verzoek toegezonden door de Bibliotheek van het KNMI, postbus 201, 3730 AE De Bilt, tel. 030 - 2 206 855, fax. 030 - 2 210 407; e-mail: biblioth@knmi.nl

▼ KNMI-PUBLICATIE MET NUMMER

150-28 Sneeuwdek in Nederland 1961-1990 / A.M.G. Klein Tank
180a List of acronyms in environmental sciences : revised edition / [compiled by P. Geerders and M. Waterborg]
181b FM12 SYNOP . internationale en nationale regelgeving voor het coderen van de groepen 7wwW1W2 en 960ww; derde druk
183-1 Rainfall in New Guinea (Irian Jaya) / T.B. Ridder
183-2 Vergelijking van zware regens te Hollandia (Nieuw Guinea), thans Jayapura (Irian Jaya) met zware regens te De Bilt / T. B. Ridder
183-3 Verdamping in Nieuw-Guinea, vergelijking van gemeten hoeveelheden met berekende hoeveelheden / T.B. Ridder
183-4 Beschrijving van het klimaat te Merauke, Nieuw Guinea, in verband met de eventuele vestiging van een zoutwinningsbedrijf / T.B. Ridder a.o.
183-5 Overzicht van klimatologische en geofysische publikaties betreffende Nieuw-Guinea / T.B. Ridder
184a Inleiding tot de algemene meteorologie : studie-uitgave ; 2e druk / B. Zwart, A. Steenhuisen, m.m.v. H.J. Krijnen
185a Handleiding voor het gebruik van sectie 2 van de FM 13-X SHIP-code voor waarnemers op zee / KNMI; KLu; KM
186-I Rainfall generator for the Rhine Basin: single-site generation of weather variables by nearest-neighbour resampling / T. Brandsma a.o.
187 De wind in de rug: KNMI-weerman schaatst de Elfstedentocht / H. van Dorp
188 SODA workshop on chemical data assimilation: proceedings; 9-10 December 1998, KNMI, De Bilt, The Netherlands

▼ TECHNISCH RAPPORT = TECHNICAL REPORT (TR)

170 DARR-94 / C.P.G. Lomme
171 EFEDA-91: documentation of measurements obtained by KNMI / W.A.A. Monna a.o.
172 Cloud lidar research at the Royal Netherlands Meteorological Institute KNMI2B2, version 2 cloud lidar analysis / A.Y. Fong a.o.
173 Measurement of the structure parameter of vertical wind-velocity in the atmospheric boundary layer / R. van der Ploeg
174 Report of the ASGASEX'94 workshop / ed. by W.A. Oost
175 Over slecht zicht, bewolking, windstoten en gladheid / J. Terpstra
176 Verification of the WAQUA/CSM-16 model for the winters 1992-93 and 1993-94 / J.W. de Vries
177 Nauwkeuriger nettostraling meten / M.K. van der Molen en W. Kohsiek
178 Neerslag in het stroomgebied van de Maas in januari 1995: waarnemingen en verificatie van modelprognoses / R.Jilderda a.o.
179 First field experience with 600PA phased array sodar / H. Klein Baltink
180 Een Kalman-correctieschema voor de wegdektemperatuurverwachtingen van het VAISALA-model / A. Jacobs
181 Calibration study of the K-Gill propeller vane / Marcel Bottema
182 Ontwikkeling van een spectraal UV-meetinstrument / Frank Helderman
183 Rainfall generator for the Rhine catchment : a feasibility study / T. Adri Buishand and Theo Brandsma
184 Parametrisatie van mooi-weer cumulus / M.C. van Zanten
185 Interim report on the KNMI contributions to the second phase of the AERO-project / Wiel Wauben, Paul Fortuin a.o.
186 Seismische analyse van de aardbevingen bij Middelstum (30 juli 1994) en Annen (16 augustus '94 en 31 januari '95) / [SO]
187 Analyse wenselijkheid overname RIVM-windmeetlokaties door KNMI / H. Benschop
188 Windsnelheidsmetingen op zeestations en kuststations: herleiding waarden windsnelheden naar 10-meter niveau / H. Benschop
189 On the KNMI calibration of net radiometers / W. Kohsiek
190 NEDWAM statistics over the period October 1994 - April 1995 / F.B. Koek
191 Description and verification of the HIRLAM trajectory model / E. de Bruijn
192 Tiltmeting : een alternatief voor waterpassing ? / H.W. Haak
193 Error modelling of scatterometer, in-situ and ECMWF model winds; a calibration refinement / Ad Stoffelen
194 KNMI contribution to the European project POPSICLE / Theo Brandsma a.o.
195 ECBILT . a coupled atmosphere ocean sea-ice model for climate predictability studies / R.J. Haarsma a.o.
196 Environmental and climatic consequences of aviation: final report of the KNMI contributions to the AERO-project / W. Wauben a.o.
197 Global radiation measurements in the operational KNMI meteorological network: effects of pollution and ventilation / F. Kuik
198 KALCORR: a kalman-correction model for real-time road surface temperature forecasting / A. Jacobs
199 Macroseismische waarnemingen Roswinkel 19-2-1997 / B. Dost e.a.
200 Operationele UV-metingen bij het KNMI / F. Kuik
201 Vergelijking van de Vaisala's HMP233 en HMP243 relatieve luchtvochtigheidsmeters / F. Kuik
202 Statistical guidance for the North Sea / Janet Wijngaard and Kees Kok
203 UV-intercomparison SUSPEN / Foeke Kuik and Wiel Wauben
204 Temperature corrections on radiation measurements using Modtran 3 / D.A. Bunskoek, A.C.A.P. van Lammeren and A.J. Feijt
205 Seismisch risico in Noord-Nederland / Th. De Crook, H.W. Haak en B. Dost
206 The HIRLAM-STAT-archive and its application programs / Albert Jacobs
207 Retrieval of aerosol properties from multispectral direct sun measurements / O.P. Hasekamp
208 The KNMI Garderen Experiment, micro-meteorological observations 1988-1989; instruments and data / F.C. Bosveld a.o.
209 CO2 in water and air during ASGAMAGE: concentration measurements and consensus data / Cor M.J. Jacobs, Gerard J. Kunz, Detlev Sprung a.o.
210 Elf jaar Cabauw-metingen / J.G. van der Vliet
211 Indices die de variabiliteit en de extremen van het klimaat beschrijven / E.J. Klok
212 First guess TAF-FGTAF: semi-automation in TAF production / Albert Jacobs
213 Zeer korte termijn bewolkingsverwachting met behulp van METCAST: een verificatie en beschrijving model-uitvoer / S.H. van der Veen
214 The implementation of two mixed-layer schemes in the HOPE ocean general circulation model / M. van Eijk
215 Stratosphere-troposphere exchange of ozone, diagnosed from an ECMWF ozone simulation experiment / Harm Luykx
216 Evaluatierapport Automatisering Visuele Waarnemingen Ontwikkeling Meestsystemen / Wiel Wauben en Hans de Jongh
217 Verificatie TAF en TREND / Hans van Bruggen
218 LEO - LSG and ECBILT coupled through OASIS: description and manual/A. Sterl
219 De invloed van de grondwaterstand, wind, temperatuur en dauwpunt op de vorming van stralingsmist: een kwantitatieve benadering / Jan Terpstra
220 Back-up modellering van windmeetmasten op luchthavens / Ilja Smits
221 PV-mixing around the tropopause in an extratropical cyclone / M. Sigmond
222 NPK-TIG oefendag 16 december 1998 / G.T. Geertsema, H. van Dorp e.a.
223 Golfhoogteverwachtingen voor de Zuidelijke Noordzee: een korte vergelijking van het ECMWF-golfmodel (EPS en operationeel), de nautische gidsverwachting, Nedwam en meteoroloog / D.H.P Vogelezang en C.J. Kok.

▼ WETENSCHAPPELIJK RAPPORT = SCIENTIFIC REPORT (WR)

96-01 A new algorithm for total ozone retrieval from direct sun measurements with a filter instrument / W.M.F. Wauben
96-02 Chaos and coupling: a coupled atmosphere ocean-boxmodel for coupled behaviour studies / G. Zondervan
96-03 An acoustical array for subsonic signals / H.W. Haak
96-04 Transformation of wind in the coastal zone / V.N. Kudryavtsev a.o.
96-05 Simulations of the response of the ocean waves in the North Atlantic and North Sea to CO2 doubling in the atmosphere / K. Rider a.o.
96-06 Microbarograph systems for the infrasonic detection of nuclear explosions / H.W. Haak and G.J. de Wilde
96-07 An ozone climatology based on ozonesonde measurements / J.P.F. Fortuin
96-08 GOME validation at KNMI and collaborating institutes / ed. P. Stammes a.o.
97-01 The adjoint of the WAM model / H. Hersbach
97-02 Optimal interpolation of partitions: a data assimilation scheme for NEDWAM-4; description and evaluation of the period November 1995 - October 1996 / A.Voorrips
97-03 SATVIEW: a semi-physical scatterometer algorithm / J.A.M. Janssen a.o.
97-04 GPS water vapour meteorology status report / H. Derks a.o.
97-05 Climatological spinup of the ECBILT oceanmodel / Arie Kattenberg a.o.
97-06 Direct determination of the air-sea transfer velocity of CO2 during ASGAMAGE / J.C.M. Jacobs, W. Kohsiek and W.A. Oost
97-07 Scattering matrices of ice crystals / M. Hess, P. Stammes a.o.
97-08 Experiments with horizontal diffusion and advection in a nested fine mesh mesoscale model / E.I.F. de Bruijn
97-09 On the assimilation of ozone into an atmospheric model / E.Valur Hólm
98-01 Steady state analysis of a coupled atmosphere ocean-boxmodel / F.A. Bakker
98-02 The ASGAMAGE workshop, September 22-25, 1997 / ed. W.A. Oost
98-03 Experimenting with a similarity measure for atmospheric flows / R.A. Pasmanter and X.-L. Wang
98-04 Evaluation of a radio interferometry lightning positioning system / H.R.A. Wessels
98-05 Literature study of climate effects of contrails caused by aircraft emissions / V.E. Pultau
99-01 Enhancement of solar and ultraviolet surface irradiance under partial cloudy conditions / Serdal Tunç
99-02 Turbulent air flow over sea waves: simplified model for applications / V.N. Kudryavtsev, V.K. Makin and J.F. Meirink
99-03 The KNMI Garderen experiment, micro-meteorological observations 1988-1989: corrections / Fred C. Bosveld
99-04 ASGAMAGE: the ASGASEX MAGE experiment . final report / ed. W.A.Oost