IDL

Scientific Data Formats

RSI

IDL Version 6.3
April 2006 Edition

0406IDL63SDF

Acknowledgments

IDL® is a registered trademark and ION™, ION Script™, ION Java™, are trademarks of ITT Industries, registered in the United States Patent and Trademark Office, for the computer program described herein.

Numerical Recipes™ is a trademark of Numerical Recipes Software. Numerical Recipes routines are used by permission.

GRG2™ is a trademark of Windward Technologies, Inc. The GRG2 software for nonlinear optimization is used by permission.

NCSA Hierarchical Data Format (HDF) Software Library and Utilities
Copyright 1988-2001 The Board of Trustees of the University of Illinois
All rights reserved.

NCSA HDF5 (Hierarchical Data Format 5) Software Library and Utilities
Copyright 1998-2002 by the Board of Trustees of the University of Illinois. All rights reserved.

CDF Library
Copyright © 2002 National Space Science Data Center
NASA/Goddard Space Flight Center

NetCDF Library
Copyright © 1993-1999 University Corporation for Atmospheric Research/Unidata

HDF EOS Library
Copyright © 1996 Hughes and Applied Research Corporation

This software is based in part on the work of the Independent JPEG Group.

Portions of this software are copyrighted by DataDirect Technologies, 1991-2003.

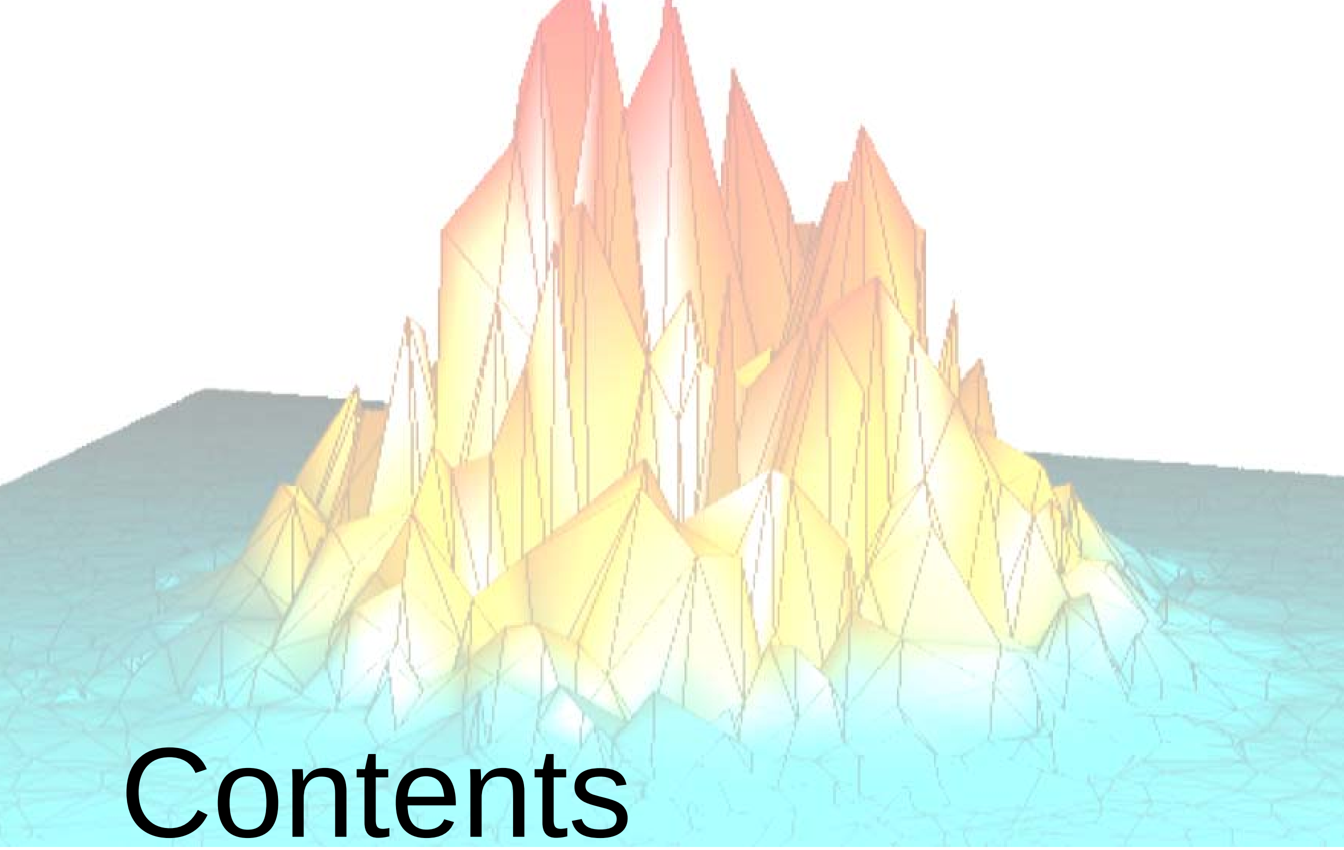
Portions of this software were developed using Unisearch's Kakadu software, for which Kodak has a commercial license. Kakadu Software. Copyright © 2001. The University of New South Wales, UNSW, Sydney NSW 2052, Australia, and Unisearch Ltd, Australia.

Portions of this computer program are copyright © 1995-1999 LizardTech, Inc. All rights reserved. MrSID is protected by U.S. Patent No. 5,710,835. Foreign Patents Pending.

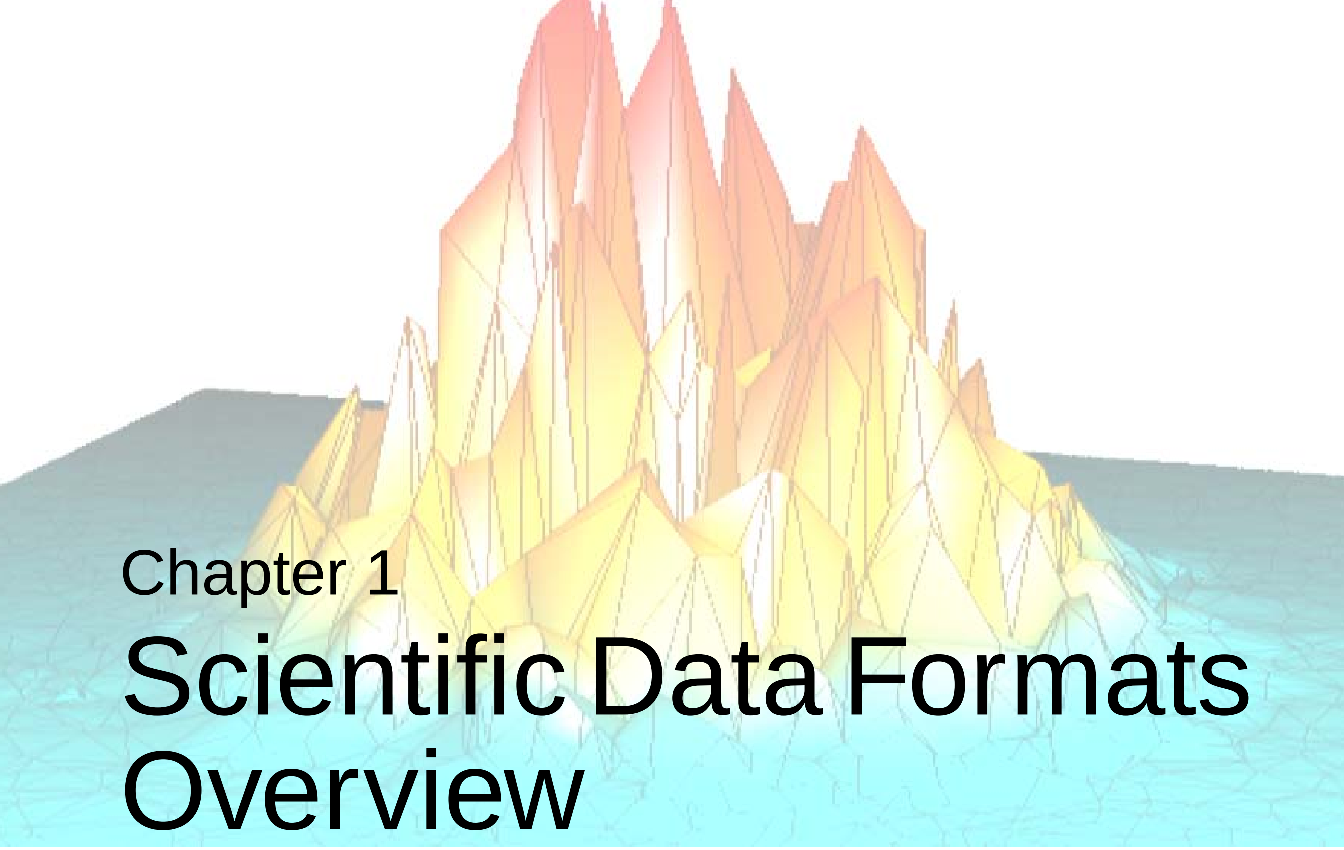
Portions of this software are copyrighted by Merge Technologies Incorporated.

IDL Wavelet Toolkit Copyright © 2002 Christopher Torrence.

Other trademarks and registered trademarks are the property of the respective trademark holders.



Contents



Chapter 1

Scientific Data Formats Overview

This chapter provides an overview the four self-describing scientific data formats supported by IDL: CDF (Common Data Format), HDF and HDF5 (Hierarchical Data Format), HDF-EOS (Earth Observing System extensions to HDF), and netCDF (Network Common Data Format). These data formats are supported on all IDL platforms. Detailed documentation for each routine can be found in this volume.

CDF—Common Data Format

The Common Data Format is a file format that facilitates the storage and retrieval of multi-dimensional scientific data. This version of IDL supports CDF 3.1. IDL's CDF routines all begin with the prefix "CDF_".

CDF is a product of the National Space Science Data Center (NSSDC). General information about CDF, including the "frequently-asked-questions" (FAQ) list, software, and CDF's IDL library (an alternative interface between CDF and IDL) are available on the World Wide Web at:

`http://nssdc.gsfc.nasa.gov/cdf/cdf_home.html`

If you do not have access to the WWW you can get CDF information via ftp at:

`ftp://nssdc.gsfc.nasa.gov/pub/cdf/FAQ.doc`

For assistance via e-mail, send a message to the internet address:

`cdfsupport@nssdca.gsfc.nasa.gov`

HDF—Hierarchical Data Format

The Hierarchical Data Format (HDF) is a multi-object file format that facilitates the transfer of various types of data between machines and operating systems. HDF is a product of the National Center for Supercomputing Applications (NCSA). HDF is designed to be flexible, portable, self-describing and easily extensible for future enhancements or compatibility with other standard formats. The HDF library contains interfaces for storing and retrieving images and multi-dimensional scientific data.

IDL supports two distinct versions of HDF: version 4 and version 5.

HDF Version 4 Support

This version of IDL supports HDF 4.1r5. IDL's HDF version 4 routines all begin with the prefix "HDF_".

HDF Version 5 Support

This version of IDL supports HDF5 5-1.6.3. IDL's HDF version 5 routines all begin with the prefix "H5_" or "H5*_".

Further information about HDF and HDF5 can be found on the World Wide Web at the HDF Information Server:

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

Alternately, you can send e-mail to `hdfhelp@ncsa.uiuc.edu`.

HDF-EOS—Hierarchical Data Format - Earth Observing System

HDF-EOS (Hierarchical Data Format-Earth Observing System) is an extension of NCSA (National Center for Supercomputing Applications) HDF and uses HDF calls as an underlying basis. This API contains functionality for creating, accessing and manipulating Grid, Point and Swath structures. IDL's HDF-EOS routines all begin with the prefix "EOS_". This version of IDL supports HDF-EOS 2.8.

HDF-EOS is a product of NASA, information may be found at:

`http://hdfeos.gsfc.nasa.gov`

NetCDF—Network Common Data Format

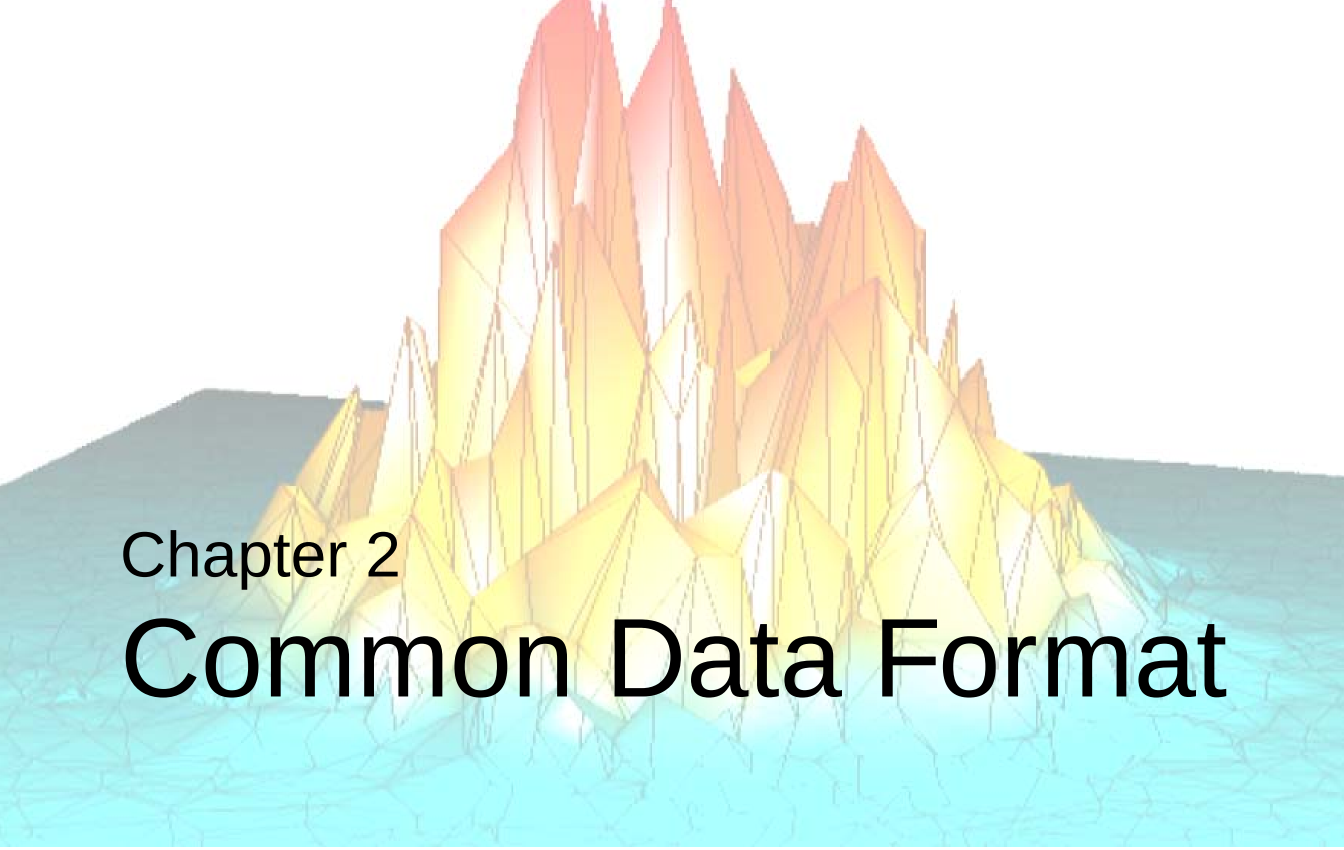
The network Common Data Format (netCDF) is a self-describing scientific data access interface and library developed at the Unidata Program Center in Boulder, Colorado. The netCDF interface and library use XDR (eXternal Data Representation) to make the data format machine-independent. This version of IDL supports netCDF 3.5. IDL's NetCDF routines all begin with the prefix "NCDF_".

More information about netCDF can be found on Unidata's netCDF World Wide Web home page which can be found at:

`http://www.unidata.ucar.edu/packages/netcdf/`

Further information and the original netCDF documentation can be obtained from Unidata at the following addresses:

UCAR Unidata Program Center
P.O. Box 3000
Boulder, Colorado, USA 80307
(303) 497-8644
e-mail: `support@unidata.ucar.edu`



Chapter 2
Common Data Format

The following topics are covered in this appendix:

Overview of the Common Data Format

The Common Data Format is a file format that facilitates the storage and retrieval of multi-dimensional scientific data. This version of IDL supports version 3.1 of the CDF library. IDL's CDF routines all begin with the prefix "CDF_".

CDF is a product of the National Space Science Data Center (NSSDC). General information about CDF, including the "frequently-asked-questions" (FAQ) list, software, and CDF's IDL library (an alternative interface between CDF and IDL) are available on the World Wide Web at:

```
http://cdf.gsfc.nasa.gov/
```

CDF documentation, including the *CDF User's Guide*, is available at:

```
http://cdf.gsfc.nasa.gov/html/docs.html
```

For assistance via e-mail, send a message to the internet address:

```
cdfsupport@listserv.gsfc.nasa.gov
```

Variables and Attributes

Information in a CDF file consists of attributes (metadata) and collections of data records (variables).

Variables

IDL can create CDF files representing any data that can be stored in a zero- to eight-dimensional array. CDF supports two distinct types of variables, *rVariables* and *zVariables*. For reasons of efficiency, CDF uses variances to indicate whether data is unique between records and dimensions. For example, consider a data set of simultaneous surface temperatures at a variety of locations, the IDL code for creating the CDF file is included at the end of this section. A variable representing “GMT time” will vary from record to record, but not dimension to dimension (since all data are taken simultaneously). On the other hand, a variable such as longitude may not vary from record to record, but will vary from dimension to dimension. Record variance is set using the REC_VARY and REC_NOVARY keywords to CDF_VARCREATE, while dimensional variance is set through the *DimVary* argument to CDF_VARCREATE. In both cases, the default is varying data.

rVariables

rVariables (or regular variables) are multidimensional arrays of values, each having the same dimensions. That is, all *rVariables* in a CDF must have the same number of dimensions and dimension sizes. In IDL, the *rVariable* dimension sizes are declared when the CDF file is first created with CDF_CREATE. In the example at the end of this section, all variables except time are *rVariables*.

zVariables

zVariables (The *z* doesn’t stand for anything—the CDF people just like the letter z) are multidimensional arrays of values of the same data type. *zVariables* can have different dimensionality from other *zVariables* and *rVariables*. In general, *zVariables* are much more flexible, and therefore easier to use, than *rVariables*.

For more discussion on CDF variables, see “Organizing Your Data in CDF” in the *CDF User’s Guide*.

Attributes

Attributes can contain auxiliary information about an entire CDF file (*global scope* attributes or gAttributes), or about particular CDF variables (*variable scope* attributes or *rAttributes*/*zAttributes* depending on variable type). CDF attributes can be scalar or vector in nature, and of any valid datatype. In the case of vector, or multiple entry, attributes the user must keep track of the entry numbers (in CDF terms these are the gEntry, rEntry, or zEntry numbers depending on attribute type). For example, every rVariable in a CDF file might have an rAttribute named “Date”. A vector *zVariable* might have a *zAttribute* named “Location” with values such as [“Melbourne Beach”, “Crowley”,...]. A global attribute “MODS” might be used to keep track of the modification history of a CDF file (see “CDF_ATTPUT” on page 43). Note however, that variables cannot have multiple attributes with the same names. In IDL, CDF attributes are created with CDF_ATTPUT and retrieved with CDF_ATTGET. For more on CDF variables, see “Attributes” in the *CDF User’s Guide*.

Specifying Attributes and Variables

Variables and attributes can be referred to either by name or by their ID numbers in most CDF routines. For example, in the CDF_VARCREATE command shown in the example under “Type Conversion” on page 29, the following command would have been equivalent:

```
; Reference by variable ID:
CDF_VARCREATE, fileid, varid, '12'
```

CDF File Options

File Type

The SINGLE_FILE and MULTI_FILE keywords to CDF_CREATE allow CDFs to be written as either:

1. all data in a single file, or
2. a separate file for each variable, plus a master file for global information.

The default is MULTI_FILE. For more discussion on CDF file format options, see “File Format Options” in the *CDF User’s Guide*.

Data Encodings/Decodings

Keywords to CDF_CREATE allow files to be written in a variety of data encoding and decoding options. (For example, the /SUN_ENCODING keyword creates a file in the SUN native encoding scheme). The default encoding/decoding is network (XDR). All CDF encodings and decodings can be written or read on all platforms, but matching the encoding with the architecture used provides the best performance. If you work in a single-platform environment most of the time, select HOST_ENCODING for maximum performance. If you know that the CDF file will be transported to a computer using another architecture, specify the encoding for the target architecture or specify NETWORK_ENCODING (the default). Specifying the target architecture provides maximum performance on that architecture; specifying NETWORK_ENCODING provides maximum flexibility.

For more on CDF encoding/decoding methods and combinations, see “Encoding” and “Decoding” in the *CDF User’s Guide*.

Creating CDF Files

The following list details the basic IDL commands needed to create a new CDF file:

- CDF_CREATE: Call this procedure to begin creating a new file. CDF_CREATE contains a number of keywords which affect the internal format of the new CDF file.
- CDF_VARCREATE: Define the variables to be used in the file.
- CDF_ATTPUT: Optionally, use attributes to describe the data.
- CDF_VARPUT: Write the appropriate data to the CDF file.
- CDF_CLOSE: Close the file.

Note

On Windows, CDF routines can save and retrieve data sets greater than 64 KB in size.

Reading CDF Files

The following commands are the basic commands needed to read data from a CDF file:

- CDF_OPEN: Open an existing CDF file.
- CDF_INQUIRE: Call this function to find the general information about the contents of the CDF file.
- CDF_CONTROL: Call this function to obtain further information about the CDF file
- CDF_VARINQ: Retrieve the names, types, sizes, and other information about the variables in the CDF file.
- CDF_VARGET: Retrieve the variable values.
- CDF_ATTINQ: Optionally, retrieve the names, scope and other information about the CDFs attributes.
- CDF_ATTGET: Optionally, retrieve the attributes.
- CDF_CLOSE: Close the file.

- If the structure of the CDF file is already known, the inquiry routines do not need to be called—only CDF_OPEN, CDF_ATTGET, CDF_VARGET, and CDF_CLOSE would be needed.

Type Conversion

Values are converted to the appropriate type before being written to a CDF file. For example, in the commands below, IDL converts the string “12” to a floating-point 12.0 before writing it:

```
varid=CDF_VARCREATE(fileid, 'VarName',['VARY','VARY'],$
   DIM=[2,3+5],/CDF_FLOAT)
CDF_VARPUT, fileid, 'VarName', '12' ; Reference by variable ID
```

Example: Creating a CDF File

The following is a simple example demonstrates the basic procedure used in creating a CDF file. See “Variables and Attributes” on page 25 for a discussion of the variances used in this example. See the documentation for individual CDF routines for more specific examples.

```
id = CDF_CREATE('Temperature.cdf', [2,3], /CLOBBER )
att_id = CDF_ATTCREATE(id, 'Title', /GLOBAL)
CDF_ATTPUT, id, att_id, 0, 'My Fancy CDF'
att1_id = CDF_ATTCREATE(id, 'Planet', /GLOBAL)
CDF_ATTPUT, id, 'Planet', 0, 'Mars'
time_id = CDF_VARCREATE(id, 'Time', ['NOVARY', 'NOVARY'], $
   /REC_VARY)
att2_id = CDF_ATTCREATE(id, 'Time Standard', /VARIABLE_SCOPE)
; times are every half hour starting a 8 am GMT.
CDF_ATTPUT, id, att2_id, time_id, 'GMT'
FOR I=0,9 DO CDF_VARPUT, id, time_id, 8.+ 0.5 * I, rec_start=I
temp_id = CDF_VARCREATE(id, 'Temp', ['VARY', 'VARY'], $
   /REC_VARY, /ZVAR, DIMENSIONS=[2,3])
long_id = CDF_VARCREATE(id, 'Longitude', ['VARY', 'VARY'], $
   /REC_NOVARY)
lat_id = CDF_VARCREATE(id, 'Latitude', ['VARY', 'VARY'], $
   /REC_NOVARY)
; write 10 temperature records:
CDF_VARPUT, id, temp_id, FINDGEN(2, 3, 10)
; create longitudes:
CDF_VARPUT, id, long_id, [[10.0, 12.0], [8.0, 6.0], [3.0, 2.0]]
; create latitudes:
CDF_VARPUT, id, lat_id, [[40.0, 42.0], [38.0, 34.0],[30.0, 31.0]]
CDF_CLOSE, id
```

Alphabetical Listing of CDF Routines

CDF_ATTCREATE

CDF_ATTDELETE

CDF_ATTEXISTS

CDF_ATTGET

CDF_ATTINQ

CDF_ATTNUM

CDF_ATTPUT

CDF_ATTRENAME

CDF_CLOSE

CDF_COMPRESSION

CDF_CONTROL

CDF_CREATE

CDF_DELETE

CDF_DOC

CDF_ENCODE_EPOCH

CDF_ENCODE_EPOCH16

CDF_EPOCH

CDF_EPOCH16

CDF_ERROR

CDF_EXISTS

CDF_INQUIRE

CDF_LIB_INFO

CDF_OPEN

CDF_PARSE_EPOCH

CDF_PARSE_EPOCH16

CDF_SET_CDF27_BACKWARD_COMPATIBLE

CDF_ATTCREATE

The CDF_ATTCREATE function creates a new attribute in the specified Common Data Format file. If successful, the attribute ID is returned.

Syntax

Result = CDF_ATTCREATE(*Id*, *Attribute_Name* [, /GLOBAL_SCOPE] [, /VARIABLE_SCOPE])

Return Value

Returns the attribute ID.

Arguments

Id

The CDF ID of the file for which a new attribute is created, returned from a previous call to CDF_OPEN or CDF_CREATE.

Attribute_Name

A string containing the name of the attribute to be created.

Keywords

GLOBAL_SCOPE

Set this keyword to make the scope of the attribute global. This is the default.

VARIABLE_SCOPE

Set this keyword to indicate that the attribute's scope is per variable.

Examples

```
id = CDF_OPEN('test') ; Create a CDF file.
xx = CDF_ATTCREATE(id, 'Attribute-1', /GLOBAL_SCOPE)
CDF_ATTRENAME, id, 'Attribute-1', 'My Favorite Attribute'
PRINT, CDF_ATTNUM(id, 'My Favorite Attribute')
CDF_CLOSE, id ; Close the CDF file.
```

Version History

Pre 4.0	Introduced

CDF_ATTDELETE

The CDF_ATTDELETE procedure deletes an attribute from the specified CDF file. Note that the attribute's entries are also deleted, and that the attributes that numerically follow the deleted attribute within the CDF file are automatically renumbered.

Syntax

CDF_ATTDELETE, *Id*, *Attribute* [, *EntryNum*] [, /ZVARIABLE]

Arguments

ID

The CDF ID of the file containing the *Attribute* to be deleted, returned from a previous call to CDF_OPEN or CDF_CREATE.

Attribute

A string containing the name or zero-based attribute number of the attribute to be deleted.

EntryNum

The entry number to delete. If EntryNum is not specified, the entire attribute is deleted. If the attribute is variable in scope, this is either the name or number of the variable the attribute is to be associated with. If the attribute is global in scope, this is the actual gEntry. It is the user's responsibility to keep track of valid gEntry numbers. Normally, gEntry numbers will begin with 0 or 1 and will increase up to MAXGENTRY (as reported in the GET_ATTR_INFO structure returned by CDF_CONTROL), but this is not required.

Keywords

ZVARIABLE

If EntryNum is a variable ID (as opposed to a variable name) and the variable is a zVariable, set this flag to indicate that the variable ID is a zVariable ID. The default is to assume that EntryNum is an rVariable ID. Note: the attribute must have a scope of VARIABLE_SCOPE.

Examples

```
cid = CDF_CREATE('DEMOattdelete')
attr1_id = CDF_ATTCREATE(cid, 'GLOBAL_ATTR1', /GLOBAL_SCOPE)
attr2_id = CDF_ATTCREATE(cid, 'GLOBAL_ATTR2', /GLOBAL_SCOPE)
attr3_id = CDF_ATTCREATE(cid, 'VAR_ATTR1', /VARIABLE_SCOPE)
attr4_id = CDF_ATTCREATE(cid, 'VAR_ATTR2', /VARIABLE_SCOPE)

; Check the number of attributes:
info = CDF_INQUIRE(cid)
HELP, info.natts

; Delete the first and third attributes:
CDF_ATTDELETE, cid, 'GLOBAL_ATTR1'
; The attribute numbers are zero-based and automatically
; renumbered
CDF_ATTDELETE, cid, 1

; Select the new first attribute:
CDF_ATTINQ, cid, 0, name, scope, MaxEntry, MaxZentry
HELP, name, scope

CDF_DELETE, cid
```

IDL Output

```
<Expression>    LONG      =            4

NAME            STRING    = 'GLOBAL_ATTR2'
SCOPE           STRING    = 'GLOBAL_SCOPE'
```

Version History

4.0.1b	Introduced

See Also

CDF_ATTCREATE, CDF_ATTGET, CDF_ATTEXISTS, CDF_ATTINQ, CDF_ATTPUT, CDF_ATTRENAME

CDF_ATTEXISTS

The CDF_ATTEXISTS function determines whether a given attribute exists in the specified CDF file. Attributes may be specified by name or number.

Syntax

Result = CDF_ATTEXISTS(*Id*, *Attribute* [, *EntryNum*] [, /ZVARIABLE])

Return Value

Returns TRUE (1) if the specified attribute exists or FALSE (0) if it does not exist

Arguments

Id

The CDF ID of the file containing the Attribute to be checked, returned from a previous call to CDF_OPEN or CDF_CREATE.

Attribute

A string containing the name or zero-based attribute number of the attribute to be checked.

EntryNum

The entry number to confirm. If EntryNum is not specified, the entire file is searched for the specified attribute. If the attribute is variable in scope, this is either the name or number of the variable the attribute is to be associated with. If the attribute is global in scope, this is the actual gEntry. It is the user's responsibility to keep track of valid gEntry numbers. Normally gEntry numbers will begin with 0 or 1 and will increase up to MAXGENTRY (as reported in the GET_ATTR_INFO structure returned by CDF_CONTROL), but this is not required.

Keywords

ZVARIABLE

If EntryNum is a variable ID (as opposed to a variable name) and the variable is a zVariable, set this flag to indicate that the variable ID is a zVariable ID. The default is to assume that EntryNum is an rVariable ID. Note: the attribute must have a scope of VARIABLE_SCOPE.

Examples

Create a function to test an attribute's existence and return a string:

```
FUNCTION exists, cdfid, attname_or_number
IF CDF_ATTEXISTS(cdfid, attname_or_number) THEN $
   RETURN,' Attribute Exists' ELSE $
   RETURN,' Attribute Does Not Exist'
END

; Create a CDF with 2 attributes:
cdfid = CDF_CREATE('DEMOattexists')
attr1_id = CDF_ATTCREATE(cdfid, 'GLOBAL_ATT' , /GLOBAL_SCOPE)
attr2_id = CDF_ATTCREATE(cdfid, 'VARIABLE_ATT', /VARIABLE_SCOPE)

; Check the existence of the two attributes, plus a third that
; does not exist:
PRINT, EXISTS(cdfid, attr1_id)
PRINT, EXISTS(cdfid, 1)
PRINT, EXISTS(cdfid, 'BAD ATTR')

CDF_DELETE, cdfid
```

IDL Output

```
Attribute Exists
Attribute Exists
Attribute Does Not Exist
```

Version History

4.0.1b	Introduced

See Also

CDF_ATTCREATE, CDF_ATTGET, CDF_ATTDELETE, CDF_ATTINQ, CDF_ATTPUT, CDF_ATTRENAME

CDF_ATTGET

The CDF_ATTGET procedure reads an attribute entry from a CDF file.

Syntax

CDF_ATTGET, *Id*, *Attribute*, *EntryNum*, *Value* [, CDF_TYPE= *variable*]
[, /ZVARIABLE]

Arguments

Id

The CDF ID, returned from a previous call to CDF_OPEN or CDF_CREATE.

Attribute

A string containing the name of the attribute or the attribute number to be written.

EntryNum

The entry number. If the attribute is variable in scope, this is either the name or number of the variable the attribute is to be associated with. If the attribute is global in scope, this is the actual gEntry. It is the user's responsibility to keep track of valid gEntry numbers. Normally, gEntry numbers will begin with 0 or 1 and will increase up to MAXGENTRY (as reported in the GET_ATTR_INFO structure returned by CDF_CONTROL), but this is not required.

Value

A named variable in which the value of the attribute is returned.

Keywords

CDF_TYPE

Set this keyword equal to a named variable that will contain the CDF type of the attribute entry, returned as a scalar string. Possible returned values are: CDF_CHAR, CDF_UCHAR, CDF_INT1, CDF_BYTE, CDF_UINT1, CDF_UINT2, CDF_INT2, CDF_UINT4, CDF_INT4, CDF_REAL4, CDF_FLOAT, CDF_REAL8, CDF_DOUBLE, CDF_EPOCH, or CDF_EPOCH16. If the type cannot be determined, "UNKNOWN" is returned.

ZVARIABLE

If EntryNum is a variable ID (as opposed to a variable name) and the variable is a zVariable, set this flag to indicate that the variable ID is a zVariable ID. The default is to assume that EntryNum is an rVariable ID.

Note

The attribute must have a scope of VARIABLE_SCOPE.

Examples

```
; Open the CDF file created in the CDF_ATTPUT example:
id = CDF_OPEN('foo')
CDF_ATTGET, id, 'Attribute2', 'Var2', x
PRINT, X, FORMAT='("[",9(X,F3.1,","),X,F3.1,"]")'
CDF_CLOSE, id ; Close the CDF file.
```

IDL Output

```
[ 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

This is the expected output, since this attribute was created with a call to FINDGEN.

Version History

Pre 4.0	Introduced
6.3	Add support for EPOCH_16 type

CDF_ATTINQ

The CDF_ATTINQ procedure obtains information about a specified attribute in a Common Data Format file.

Syntax

CDF_ATTINQ, *Id*, *Attribute*, *Name*, *Scope*, *MaxEntry* [, *MaxZEntry*]

Arguments

Id

The CDF ID of the file containing the desired attribute, returned from a previous call to CDF_OPEN or CDF_CREATE.

Attribute

A string containing either the name or number of the attribute to be inquired.

Name

A named variable in which the name of the attribute is returned.

Scope

A named variable in which a string, describing the scope of the attribute, is returned. This string will have one of the following values: "GLOBAL_SCOPE", "VARIABLE_SCOPE", "GLOBAL_SCOPE_ASSUMED", or "VARIABLE_SCOPE_ASSUMED".

MaxEntry

A named variable in which the maximum rVariable entry number for this attribute is returned.

MaxZEntry

A named variable in which the maximum zVariable entry number for this attribute is returned.

Keywords

None

Examples

```
cdfid= CDF_OPEN('/cdrom/ozone.8.20.92')
CDF_ATTINQ, cdfid, 0, name, scope, maxentry, maxzentry
PRINT, name, scope, maxentry, maxzentry
```

Version History

Pre 4.0	Introduced

CDF_ATTNUM

The CDF_ATTNUM function returns the attribute number associated with a particular attribute in a Common Data Format file.

Syntax

Result = CDF_ATTNUM(*Id*, *Attribute_Name*)

Return Value

Returns the attribute number.

Arguments

Id

The CDF ID for the file that contains the desired attribute, returned from a previous call to CDF_OPEN or CDF_CREATE.

Attribute_Name

A string containing the name of the attribute.

Keywords

None

Examples

See the example for "CDF_ATTPUT" on page 43.

Version History

Pre 4.0	Introduced

CDF_ATTPUT

The CDF_ATTPUT procedure writes an attribute entry to a Common Data Format file, or attaches an attribute to a CDF variable. If the specified entry already exists, it is overwritten.

Syntax

CDF_ATTPUT, *Id*, *Attribute*, *EntryNum*, *Value* [, /ZVARIABLE]

Arguments

Id

The CDF ID, returned from a previous call to CDF_OPEN or CDF_CREATE.

Attribute

A string containing either the name or number of the attribute to be written.

EntryNum

The entry number. If the attribute is variable in scope, this is either the name or number of the variable the attribute is to be associated with. If the attribute is global in scope, this is the actual gEntry. It is the user's responsibility to keep track of valid gEntry numbers. Normally gEntry numbers will begin with 0 or 1 and will increase up to MAXGENTRY (as reported in the GET_ATTR_INFO structure returned by CDF_CONTROL), but this is not required.

Value

The value(s) to be written.

Keywords

ZVARIABLE

If EntryNum is a variable ID (as opposed to a variable name) and the variable is a zVariable, set this flag to indicate that the variable ID is a zVariable ID. The default is to assume that EntryNum is an rVariable ID. Note: the attribute must have a scope of VARIABLE_SCOPE.

Examples

Example 1

```
Id= CDF_CREATE('foo', /SUN_ENCODING, /HOST_DECODING, $
   /ROW_MAJOR)   ; no dimensions.
dummy= CDF_VARCREATE(id, 'Var1', /CDF_INT4, /REC_VARY)
v2= CDF_VARCREATE(id, 'Var2', /CDF_FLOAT, /REC_NOVARY)
dummy= CDF_ATTCREATE(id, 'Title', /VARIABLE)
global_dummy = CDF_ATTCREATE(id,'Date',/GLOBAL)
dummy= CDF_ATTCREATE(id, 'Att2', /VARIABLE)
CDF_ATTPUT, id, 'Title', 'Var1', 'Temperature at surface'
CDF_ATTPUT, id, 'Title', v2, 'Time of recording'
CDF_ATTPUT, id, 'Date',1,'July 4, 1996'
CDF_ATTPUT, id, 'Att2', 'Var2', FINDGEN(10)

; Rename the "Att2" attribute to "Attribute2":
CDF_ATTRENAME, Id, 'Att2', 'Attribute2'

; Verify the attribute number (zero-based) of Attribute2
PRINT, CDF_ATTNUM(id, 'Attribute2')

; Close the CDF file. This file is used in the CDF_ATTGET example.
CDF_CLOSE, id
```

IDL Output

```
1
```

Example 2

The following example uses the Global attribute “MODS” to keep track of the modification history of a CDF file named `mods.cdf`.

```
id = CDF_CREATE('mods.cdf', /CLOBBER)
cid = CDF_ATTCREATE(id, 'MODS', /GLOBAL_SCOPE)
CDF_ATTPUT, id, cid, 0, 'Original Version'
CDF_CLOSE, id

; Next, reopen the CDF file and make modifications:
id = CDF_OPEN('mods.cdf')
CDF_CONTROL, id, ATTRIBUTE='MODS', GET_ATTR_INFO=ginfo

;Use CDF_CONTROL to get the MAXGENTRY used.
CDF_ATTPUT, id, cid, ginfo.maxgentry+1,'Second Version'

;Insert the new gEntry at MAXGENTRY+1.
CDF_CLOSE, id
```

```
; Reopen the CDF file again and make more modifications:
id = CDF_OPEN('mods.cdf')
CDF_CONTROL, id, ATTRIBUTE='MODS', GET_ATTR_INFO=ginfo
CDF_ATTPUT, id, cid, ginfo.maxgentry+1, 'Third Version'
CDF_CLOSE, id

;Reopen the CDF file again and make a modification in the
;MAXGENTRY + 2 spot (skipping an entry number).
id = CDF_OPEN('mods.cdf')
CDF_CONTROL, id, ATTRIBUTE='MODS', GET_ATTR_INFO=ginfo
CDF_ATTPUT, id, cid, ginfo.maxgentry+2, 'Fourth Version'

; Now, examine the CDF file to review its modification history.
; Since the gENTRY numbers have a gap in them, we can check each
; attribute with the CDF_ATTEXISTS function. This is a good idea
; if you do not know for certain that the attribute entries are
; serially numbered.

CDF_CONTROL, id, ATTRIBUTE='MODS', GET_ATTR_INFO=ginfo
  FOR I=0, ginfo.maxgentry DO BEGIN
   IF CDF_ATTEXISTS(id, cid, I) THEN BEGIN
     CDF_ATTGET, id, cid, I, gatt
     PRINT, I, gatt, FORMAT='("Attribute: MODS (gENTRY #",i1,") = ",A)'
   ENDIF ELSE BEGIN
     PRINT, I, FORMAT='("Attribute: MODS (gENTRY #",i1,") $
       Does not exist")'
   ENDELSE
  ENDFOR
CDF_CLOSE, id
```

IDL Output

```
Attribute: MODS (gENTRY #0) = Original Version
Attribute: MODS (gENTRY #1) = Second Version
Attribute: MODS (gENTRY #2) = Third Version
Attribute: MODS (gENTRY #3) Does not exist
Attribute: MODS (gENTRY #4) = Fourth Version
```

Version History

Pre 4.0	Introduced

CDF_ATTRENAME

The CDF_ATTRENAME procedure is used to rename an existing attribute in a Common Data Format file.

Syntax

CDF_ATTRENAME, *Id*, *OldAttr*, *NewName*

Arguments

Id

The CDF ID of the file containing the desired attribute, returned from a previous call to CDF_OPEN or CDF_CREATE.

OldAttr

A string containing the current name of the attribute *or* the attribute number to be renamed.

NewName

A string containing the new name for the attribute.

Keywords

None

Examples

See the example for "CDF_ATTPUT" on page 43.

Version History

Pre 4.0	Introduced

CDF_CLOSE

The CDF_CLOSE procedure closes the specified Common Data Format file. The CDF's data buffers are flushed, all of the CDF's open files are closed, and the CDF identifier is freed. You *must* use CDF_CLOSE to close a CDF file to guarantee that all modifications you have made are actually written to disk.

Syntax

CDF_CLOSE, *Id*

Arguments

Id

The CDF ID of the file to be closed, returned from a previous call to CDF_OPEN or CDF_CREATE.

Keywords

None

Examples

```
; Open a file:
id = CDF_OPEN('open_close.cdf'
; ... Other CDF_ commands go here.
; Close the cdf file.
CDF_CLOSE, id
```

Version History

Pre 4.0	Introduced

CDF_COMPRESSION

The CDF_COMPRESSION procedure sets or returns the compression mode for a CDF file and/or variables. Compression parameters should be set before values are written to the CDF file.

Special Note About Temporary File Location

CDF creates temporary files whenever files/variables are compressed or uncompressed. By default, these files are created in the current directory. UNIX users can set the environment variable CDF_TMP to set the temporary directory explicitly.

Syntax

CDF_COMPRESSION, Id [, GET_COMPRESSION=*variable*]
[, GET_GZIP_LEVEL=*variable*] [, GET_VAR_COMPRESSION=*variable*]
[, GET_VAR_GZIP_LEVEL=*variable*] [, SET_COMPRESSION={0 | 1 | 2 | 3 | 5}] [, SET_GZIP_LEVEL=*integer*{1 to 9}]
[, SET_VAR_COMPRESSION={0 | 1 | 2 | 3 | 5}]
[, SET_VAR_GZIP_LEVEL=*integer*{1 to 9}]
[, VARIABLE=*variable name or index*] [, /ZVARIABLE]

Arguments

Id

The CDF ID of the file being compressed or queried, as returned from a previous call to CDF_OPEN or CDF_CREATE. Note that CDF compression only works for single-file CDF files (see CDF_CREATE).

Keywords

GET_COMPRESSION

Set this keyword to a named variable to retrieve the compression type used for the single-file CDF file. Note that individual CDF variables may compression types different that the one for the rest of the CDF file.

GET_GZIP_LEVEL

Set this keyword to a named variable in which the current GZIP effort level (1-9) for the CDF file is returned. If the compression type for the file is not GZIP (5), then a value of zero is returned.

GET_VAR_COMPRESSION

Set this keyword to a named variable to retrieve the compression type for the variable identified by the VARIABLE keyword.

GET_VAR_GZIP_LEVEL

Set this keyword to a named variable in which the GZIP effort level (1-9) for variable specified by the VARIABLE keyword is returned. If the compression type for the variable is not GZIP (5), then a value of zero is returned.

SET_COMPRESSION

Set this keyword to the compression type to be used for the single-file CDF file. Note that individual CDF variables may use compression types different than the one for the rest of the CDF file. Valid compression types are:

- 0 = No Compression
- 1 = Run-Length Encoding
- 2 = Huffman
- 3 = Adaptive Huffman
- 5 = GZIP (see the optional GZIP_LEVEL keyword)

SET_GZIP_LEVEL

This keyword is used to indicate the desired effort for the GZIP compression. This effort must be expressed as a scalar in the range (1-9). If GZIP_LEVEL is not specified upon entry then the default effort level is taken to be 5. If the SET_GZIP_LEVEL keyword is set to a valid value, and the keyword SET_COMPRESSION is not specified, the SET_COMPRESSION is set to GZIP (5).

SET_VAR_COMPRESSION

Set this keyword to the compression type for the variable identified by the VARIABLE keyword. If the variable is a zVariable, and is referred to by index in the VARIABLE keyword, then the keyword ZVARIABLE must be set. The desired variable compression should be set before variable data is added with CDF_VARPUT. Valid compression types are:

- 0 = No Compression
- 1 = Run-Length Encoding
- 2 = Huffman
- 3 = Adaptive Huffman
- 5 = GZIP (see the optional GZIP_LEVEL keyword)

SET_VAR_GZIP_LEVEL

Set this keyword to the GZIP effort level (1-9). If the compression type for the variable is not GZIP (5), no action is performed.

VARIABLE

Set this keyword to the name of a variable or a variable index to set the current variable. This keyword is mandatory when queering/setting the compression parameters of a rVariable or zVariable. Note that if VARIABLE is set to the index of a zVARIABLE, the ZVARIABLE keyword must also be set. If ZVARIABLE is not set, the variable is assumed to be an rVariable.

ZVARIABLE

Set this keyword if the current variable is a zVARIABLE and is referred to by index in the VARIABLE keyword. For example:

```
CDF_COMPRESSION, id, VARIABLE=0, /ZVARIABLE,$
   GET_VAR_COMPRESSION=vComp
```

Examples

```
; Create a  CDF file and define the compression.
; Compression only works on Single-File CDFs:
id=CDF_CREATE('demo.cdf',[10,20],/CLOBBER,/SINGLE_FILE)
CDF_COMPRESSION,id,SET_COMPRESSION=1 ; (Run-length encoding)
att_id=CDF_ATTCREATE(id, 'Date',/GLOBAL)
CDF_ATTPUT,id,'Date',att_id,systime()

; Change the compression type for the file to GZIP by using
; SET_GZIP_LEVEL:
CDF_COMPRESSION,id,SET_GZIP_LEVEL=7

; Retrieve compression information:
```

```
CDF_COMPRESSION,id,GET_GZIP_LEVEL=glevel,GET_COMPRESSION=gcomp
HELP,glevel,gcomp

; Create and compress an rVariable:
rid=CDF_VARCREATE(id,'rvar0',[1,1],/CDF_FLOAT)
CDF_COMPRESSION,id,SET_VAR_COMPRESSION=2,VARIABLE='rvar0'
CDF_VARPUT,id,'rvar0',findgen(10,20,5)
CDF_COMPRESSION,id,GET_VAR_COMPRESSION=v_comp,VARIABLE=rid,GET_VAR
_GZIP_LEVEL=v_glevel
HELP,v_comp,v_glevel

; Create and compress a  zVariable:
zid=CDF_varcreate(id,'zvar0',[1,1,1],DIM=[10,20,30],/ZVARIABLE,$
   /CDF_DOUBLE)

; You can set a compression and check it in the same call:
CDF_COMPRESSION,id,SET_VAR_GZIP_LEVEL=9,VARIABLE=zid,/ZVARIABLE,$
   GET_VAR_GZIP_LEVEL=v_gzip
HELP,v_gzip

CDF_VARPUT,id,zid,dindgen(10,20,30),/ZVARIABLE

; File and variable keywords can be combined in the same call
; (Set calls are processed before Get calls)
CDF_COMPRESSION,id,GET_VAR_COMPRESSION=v_comp,VARIABLE='zvar0',$
   /ZVARIABLE, SET_COMPRESSION=2,GET_COMPRESSION=file_comp
HELP,file_comp,v_comp

CDF_DELETE,id
```

IDL Output

```
GLEVEL          LONG      =            7
GCOMP           LONG      =            5

V_COMP          LONG      =            2
V_GLEVEL        LONG      =            0
```

(Note that V_GLEVEL is 0, since the variable compression is not GZIP.)

```
V_GZIP          LONG      =            9

FILE_COMP       LONG      =            2
V_COMP          LONG      =            5
```

Version History

5.3	Introduced

See Also

CDF_CONTROL, CDF_CREATE, CDF_OPEN, CDF_VARNUM

CDF_CONTROL

The CDF_CONTROL procedure allows you to obtain or set information for a Common Data Format file, its variables, and its attributes.

Syntax

CDF_CONTROL, *Id* [, ATTRIBUTE=*name or number*]
[, GET_ATTR_INFO=*variable*] [, GET_CACHESIZE=*variable*]
[, GET_COPYRIGHT=*variable*] [, GET_FILENAME=*variable*]
[, GET_FORMAT=*variable*] [, GET_NEGTOPOSFP0_MODE=*variable*]
[, GET_NUMATTRS=*variable*] [, GET_READONLY_MODE=*variable*]
[, GET_RVAR_CACHESIZE=*variable*] [, GET_VAR_INFO=*variable*]
[, GET_ZMODE=*variable*] [, GET_ZVAR_CACHESIZE=*variable*]
[, SET_CACHESIZE=*value*] [, SET_EXTENDRECS=*records*]
[, SET_INITIALRECS=*records*] [, /SET_NEGTOPOSFP0_MODE]
[, SET_PADVALUE=*value*] [, /SET_READONLY_MODE]
[, SET_RVAR_CACHESIZE=*value*{See Note}]
[, SET_RVARS_CACHESIZE=*value*{See Note}] [, SET_ZMODE={0 | 1 | 2}]
[, SET_ZVAR_CACHESIZE=*value*{See Note}]
[, SET_ZVARS_CACHESIZE=*value*{See Note}] [, VARIABLE=*name or index*]
[, /ZVARIABLE]

Note: Use only with MULTI_FILE CDF files

Arguments

Id

The CDF ID of the file being changed or queried, as retuned from a previous call to CDF_OPEN or CDF_CREATE.

Keywords

ATTRIBUTE

Makes the attribute specified the current attribute. Either an attribute name or an attribute number may be specified.

GET_ATTR_INFO

Set this keyword to a named variable that will contain information about the current attribute. Information is returned in the form of a structure with the following tags:

```
{ NUMGENTRIES:0L, NUMRENTRIES:0L, NUMZENTRIES:0L,
  MAXGENTRY:0L, MAXRENTRY:0L, MAXZENTRY:0L }
```

The first three tags contain the number of globals, rVariables, and zVariables associated with the attribute. MAXGENTRY contains the highest index used, and the last two tags contain the highest variable ids that were used when setting the attribute’s value.

Note that an attribute must be set before GET_ATTR_INFO can be used. For example:

```
CDF_CONTROL, id, ATTRIBUTE='ATT1', GET_ATTR_INFO=X
```

GET_CACHESIZE

Set this keyword to a named variable that will be set equal to the number of 512-byte cache buffers being used for the current `.cdf` file. For discussion about using caches with CDF files, see “Caching Scheme” in the *CDF User’s Guide*.

GET_COPYRIGHT

Set this keyword to a named variable that will contain the copyright notice of the CDF library now being used by IDL (as opposed to the library that was used to write the current CDF).

GET_FILENAME

Set this keyword to a named variable that will contain the pathname of the current `.cdf` file.

GET_FORMAT

Set this keyword to a named variable that will contain a string describing the CDF Format of the current CDF file. Possible formats are SINGLE_FILE and MULTI_FILE, and can only be set with the CDF_CREATE procedure. For example:

```
id = CDF_CREATE('single', /SINGLE_FILE)
CDF_CONTROL, id, GET_FORMAT = cdfformat
HELP, cdfformat
```

IDL prints:

```
CDFFORMAT      STRING    = 'SINGLE_FILE'
```

GET_NEGTOPOSFP0_MODE

Set this keyword to a named variable that will be set equal to the CDF negative to positive floating point 0.0 (NEGtoPOSfp0) mode. In NEGtoPOSfp0 mode, values equal to -0.0 will be converted to 0.0 whenever encountered. By CDF convention, a returned value of -1 indicates that this feature is enabled, and a returned value of zero indicates that this feature is disabled.

GET_NUMATTRS

Set this keyword to a named variable that will contain a two-element array of longs. The first value will contain the number of attributes with global scope; the second value will contain the number of attributes with variable scope. NOTE: attributes with GLOBAL_SCOPE_ASSUMED scope will be included in the global scope count and attributes with VARIABLE_SCOPE_ASSUMED will be included in the count of attributes with variable scope.

Note that you can obtain the total number of attributes using the CDF_INQUIRE routine.

GET_READONLY_MODE

Set this keyword to a named variable that will be set equal to the CDF read-only mode. By CDF convention, a returned value of -1 indicates that the file is in read-only mode, and a returned value of zero indicates that the file is not in read-only mode.

GET_RVAR_CACHESIZE

Set this keyword to a named variable that will be set equal to the number of 512-byte cache buffers being used for the current MULTI_FILE format CDF and the rVariable indicated by the VARIABLE keyword. This keyword should only be used for MULTI_FILE CDF files. For discussion about using caches with CDF files, see “Caching Scheme” in the *CDF User’s Guide*.

GET_VAR_INFO

Set this keyword to a named variable that will contain information about the current variable. For detailed information about the returned values, see “Records” in the *CDF User’s Guide*. Information is returned in the form of a structure with the following tags:

```
{ EXTENDRECS:0L, MAXALLOCREC:0L, MAXREC:0L,
   MAXRECS:0L, NINDEXENTRIES:0L, NINDEXRECORDS:0L,
   PADVALUE:<as appropriate> }
```

The EXTENDRECS field will contain the number of records by which the current variable will be extended whenever a new record needs to be added.

The MAXALLOCREC field will contain the maximum record number (zero-based) allocated for the current variable. Records can only be allocated for NOVARY zVariables in SINGLE_FILE format CDFs. When these conditions are not met, the value is set to -1.

The MAXREC field will contain the maximum record number for the current variable. For variables with a record variance of NOVARY, this will be at most zero. A value of -1 indicates that no records have been written.

The MAXRECS field will contain the maximum record number (zero-based) of all variables of this type (rVariable or zVariable) in the current CDF. A value of -1 indicates that no records have been written.

The NINDEXENTRIES field will contain the number of index entries for the current variable in the current CDF. This value is -1 unless the current CDF is of SINGLE_FILE format, and the variable is a zVariable.

The NINDEXRECORDS field will contain the number of index records for the current variable in the current CDF. This value is -1 unless the current CDF is of SINGLE_FILE format, and the variable is a zVariable.

The PADVALUE field will contain the value being used to fill locations that are not explicitly filled by the user. If a PADVALUE is not specified, CDF_CONTROL returns an error.

For example:

```
fid = CDF_CREATE('test.cdf')
varid = CDF_VARCREATE(fid, 'test')
CDF_CONTROL, fid, GET_VAR_INFO=info, VARIABLE='test'
```

IDL Prints:

```
% CDF_CONTROL: Function completed but
NO_PADVALUE_SPECIFIED: A pad value has not been specified.
```

GET_ZMODE

Set this keyword to a named variable that will be set equal the zMode of the current CDF. In a non-zero zMode, CDF rVariables are temporarily replaced with zVariables. The possible return values are:

- 0 = zMode is off.
- 1 = zMode is on in zMode/1, indicating that the dimensionality and variances of the variables will stay the same.

- 2 = zMode is on in zMode/2, indicating that those dimensions with false variances (NOVARY) will be eliminated.

For Information about zModes, see “CDF Modes” in the *CDF User’s Guide*.

GET_ZVAR_CACHESIZE

Set this keyword to a named variable that will be set equal to the number of 512-byte cache buffers being used in the current MULTI_FILE format CDF and the zVariable indicated by the VARIABLE keyword. This keyword should only be used with MULTI_FILE CDF files. For discussion about using caches with CDF files, see “Caching Scheme” in the *CDF User’s Guide*.

SET_CACHESIZE

Set this keyword equal to the desired number of 512-byte cache buffers to used for the current `.cdf` file. For discussion about using caches with CDF files, see “Caching Scheme” in the *CDF User’s Guide*.

SET_EXTENDRECS

Set this keyword equal to the number of additional physical records that should be added to the current variable whenever it needs to be extended.

SET_INITIALRECS

Set this keyword equal to the number of records that should be initially written to the current variable. Note that this keyword should be set *before* writing any data to the variable.

SET_NEGTOPOSFP0_MODE

Set this keyword to a non-zero value to put the current CDF file into negative to positive floating point 0.0 (NEGtoPOSfp0) mode. In this mode, values equal to -0.0 will be converted to 0.0 whenever encountered. Setting this keyword equal to zero takes the current CDF file out of NEGtoPOSfp0 mode.

SET_PADVALUE

Set this keyword equal to the pad value for the current variable.

SET_READONLY_MODE

Set this keyword to a non-zero value to put the current CDF file into read-only mode. Set this keyword equal to zero to take the current CDF file out of read-only mode.

SET_RVAR_CACHESIZE

Set this keyword equal to the desired number of 512-byte cache buffers to used for the rVariable file specified by the VARIABLE keyword. This keyword should only be used with MULTI_FILE CDF files. For discussion about using caches with CDF files, see “Caching Scheme” in the *CDF User’s Guide.*

SET_RVARS_CACHESIZE

Set this keyword equal to the desired number of 512-byte cache buffers to used for all rVariable files in the current CDF file or files. This keyword should only be used with MULTI_FILE CDF files. For discussion about using caches with CDF files, see “Caching Scheme” in the *CDF User’s Guide*.

SET_ZMODE

Set this keyword to change the zMode of the current CDF. In a non-zero zMode, CDF rVariables are temporarily replaced with zVariables. Set this keyword to one (1) to change to zMode/1, in which the dimensionality and variances of the variables stay the same. Set this keyword to two (2) to change to zMode/2, in which those dimensions with false variances (NOVARY) are eliminated. For Information about zModes, see “CDF Modes” in the *CDF User’s Guide*.

SET_ZVAR_CACHESIZE

Set this keyword equal to the desired number of 512-byte cache buffers to used for the zVariable’s file specified by the VARIABLE keyword. This keyword should only be used with MULTI_FILE CDF files. For discussion about using caches with CDF files, see “Caching Scheme” in the *CDF User’s Guide*.

SET_ZVARS_CACHESIZE

Set this keyword equal to the desired number of 512-byte cache buffers to used for all zVariable files in the current CDF. This keyword should only be used with MULTI_FILE CDF files. For discussion about using caches with CDF files, see “Caching Scheme” in the *CDF User’s Guide*.

VARIABLE

Set this keyword to a name or index to set the current variable. The following example specifies that the variable MyData should have 20 records written to it initially:

```
CDF_CONTROL, id, VAR='MyData', SET_INITIALRECS=20
```

Note that if VARIABLE is set to the index of a zVariable, the ZVARIABLE keyword *must* also be set. If ZVARIABLE is not set, the variable is assumed to be an rVariable.

ZVARIABLE

Set this keyword to TRUE if the current variable is a zVariable and is referred to by index. For example:

```
CDF_CONTROL, id, VARIABLE=0, /ZVARIABLE, GET_VAR_INFO=V
```

Version History

Pre 4.0	Introduced

See Also

CDF_CREATE, CDF_INQUIRE

CDF_CREATE

The CDF_CREATE function creates a new Common Data Format file with the given filename and dimensions.

Note that when you create a CDF file, you may specify both encoding and decoding methods. Encoding specifies the method used to write data to the CDF file. Decoding specifies the method used to retrieve data from the CDF file and pass it to an application (IDL, for example). Encoding and decoding methods are specified by setting the *XXX*_ENCODING and *XXX*_DECODING keywords to CDF_CREATE. If no decoding method is specified, the decoding method is set to be the same as the encoding method.

All CDF encodings and decodings can be written or read on all platforms, but matching the encoding with the architecture used provides the best performance. Since most people work in a single-platform environment most of the time, the HOST_ENCODING method is used as the default encoding scheme. This provides maximum performance. If you know that the CDF file will be transported to a computer using another architecture, specify the encoding for the target architecture or specify NETWORK_ENCODING. Specifying the target architecture provides maximum performance on that architecture; specifying NETWORK_ENCODING provides maximum flexibility.

For more discussion on CDF encoding/decoding methods and combinations, see "Encoding" and "Decoding" in the *CDF User's Guide*.

Note

Versions of IDL beginning with 6.3 support the CDF 3.1 library. If you need to create CDF files that can be read by earlier versions of IDL, or by CDF libraries earlier than version 3.0, use the CDF_SET_CDF27_BACKWARD_COMPATIBLE routine.

Syntax

Result = CDF_CREATE(*Filename*, [*Dimensions*] [, /CLOBBER] [, /MULTI_FILE | , /SINGLE_FILE] [, /COL_MAJOR | , /ROW_MAJOR])

Encoding Keywords (pick one):

[, /ALPHAOSF1_ENCODING]
[, /ALPHAVMSD_ENCODING]
[, /ALPHAVMSG_ENCODING]
[, /DECSTATION_ENCODING]

[, /HOST_ENCODING]
[, /HP_ENCODING]
[, /IBMPC_ENCODING]
[, /IBMRS_ENCODING]
[, /MAC_ENCODING]
[, /NETWORK_ENCODING]
[, /NEXT_ENCODING]
[, /SGI_ENCODING]
[, /SUN_ENCODING]

Decoding Keywords (pick one):
[, /ALPHAOSF1_DECODING]
[, /ALPHAVMSD_DECODING]
[, /ALPHAVMSG_DECODING]
[, /DECSTATION_DECODING]
[, /HOST_DECODING]
[, /HP_DECODING]
[, /IBMPC_DECODING]
[, /IBMRS_DECODING]
[, /MAC_DECODING]
[, /NETWORK_DECODING]
[, /NEXT_DECODING]
[, /SGI_DECODING]
[, /SUN_DECODING]

Return Value

Returns the CDF ID for the new file.

Arguments

Filename

A scalar string containing the name of the file to be created. Note that if the desired filename has a `.cdf` ending, you can omit the extension and specify just the first part of the filename. For example, specifying `“mydata”` would open the file `mydata.cdf`.

Dimensions

A vector of values specifying size of each rVariable dimension. If no dimensions are specified, the file will contain a single scalar per record (*i.e.*, a 0-dimensional CDF). This argument has no effect on zVariables.

Keywords

CLOBBER

Set this keyword to erase the existing file (if the file already exists) before creating the new version.

Note that if the existing file has been corrupted, the CLOBBER operation may fail, causing IDL to display an error message. In this case you must manually delete the existing file from outside IDL.

COL_MAJOR

Set this keyword to use column major (IDL-like) array ordering for variable storage.

MULTI_FILE

Set this keyword to cause all CDF control information and attribute entry data to be placed in one `.cdf` file, with a separate file created for each defined variable. If the variable is an rVariable, then the variable files will have extensions of `.v0`, `.v1`, *etc.*; zVariables will be stored in files with extensions of `.z0`, `.z1`, *etc*. See “Format” in the *CDF User’s Guide* for more information. If both SINGLE_FILE and MULTI_FILE are set the file will be created in the SINGLE_FILE format.

Note

In versions of IDL prior to 6.3, MULTI_FILE was the default.

MULTI_FILE Example:

```
id=CDF_CREATE('multi', /MULTI_FILE)
CDF_CONTROL, id, GET_FORMAT=cdf_format
HELP, cdf_format
```

IDL prints:

```
CDF_FORMAT      STRING    = 'MULTI_FILE'
```

ROW_MAJOR

Set this keyword to specify row major (C-like) array ordering for variable storage. This is the default.

SINGLE_FILE

Set this keyword to cause all CDF information (control information, attribute entry data, variable data, etc.) to be written to a single `.cdf` file. This is the default. See "Format" in the *CDF User's Guide* for more information. If both SINGLE_FILE and MULTI_FILE are set the file will be created in the SINGLE_FILE format.

Note

In versions of IDL prior to 6.3, MULTI_FILE was the default.

Encoding Keywords

Select one of the following keywords to specify the type of encoding:

ALPHAOSF1_ENCODING

Set this keyword to indicate DEC ALPHA/OSF1 data encoding.

ALPHAVMSD_ENCODING

Set this keyword to indicate DEC ALPHA/VMS data encoding using Digital's D_FLOAT representation.

ALPHAVMSG_ENCODING

Set this keyword to indicate DEC ALPHA/VMS data encoding using Digital's G_FLOAT representation.

DECSTATION_ENCODING

Set this keyword to select Decstation (MIPSEL) data encoding.

HOST_ENCODING

Set this keyword to select that the file will use native data encoding. This is the default method.

HP_ENCODING

Set this keyword to select HP 9000 data encoding.

IBMPC_ENCODING

Set this keyword to select IBM PC data encoding.

IBMRS_ENCODING

Set this keyword to select IBM RS/6000 series data encoding.

MAC_ENCODING

Set this keyword to select Macintosh data encoding.

NETWORK_ENCODING

Set this keyword to select network-transportable data encoding (XDR).

NEXT_ENCODING

Set this keyword to select NeXT data encoding.

SGI_ENCODING

Set this keyword to select SGI (MIPSEB) data encoding (Silicon Graphics Iris and Power series).

SUN_ENCODING

Set this keyword to select SUN data encoding.

Decoding Keywords

Select one of the following keywords to specify the type of decoding:

ALPHAOSF1_DECODING

Set this keyword to indicate DEC ALPHA/OSF1 data decoding.

ALPHAVMSD_DECODING

Set this keyword to indicate DEC ALPHA/VMS data decoding using Digital's D_FLOAT representation.

ALPHAVMSG_DECODING

Set this keyword to indicate DEC ALPHA/VMS data decoding using Digital's G_FLOAT representation.

DECSTATION_DECODING

Set this keyword to select Decstation (MIPSEL) data decoding.

HOST_DECODING

Set this keyword to select that the file will use native data decoding. This is the default method.

HP_DECODING

Set this keyword to select HP 9000 data decoding.

IBMPC_DECODING

Set this keyword to select IBM PC data decoding.

IBMRS_DECODING

Set this keyword to select IBM RS/6000 series data decoding.

MAC_DECODING

Set this keyword to select Macintosh data decoding.

NETWORK_DECODING

Set this keyword to select network-transportable data decoding (XDR).

NEXT_DECODING

Set this keyword to select NeXT data decoding.

SGI_DECODING

Set this keyword to select SGI (MIPSEB) data decoding (Silicon Graphics Iris and Power series).

SUN_DECODING

Set this keyword to select SUN data decoding.

Examples

Use the following command to create a 10-element by 20-element CDF using network encoding and Sun decoding:

```
id = CDF_CREATE('cdf_create.cdf', [10,20], /NETWORK_ENCODING, $
   /SUN_DECODING)
; ... other cdf commands ...
CDF_CLOSE, id ; close the file.
```

Now suppose that we decide to use HP_DECODING instead. We can use the CLOBBER keyword to delete the existing file when creating the new file:

```
id = CDF_CREATE('cdf_create.cdf', [10,20], /NETWORK_ENCODING, $
   /HP_DECODING, /CLOBBER)
; ... other cdf commands ...
CDF_CLOSE, id ; close the file.
```

The new file is written over the existing file. Use the following command to delete the file:

```
CDF_DELETE, id
```

Version History

Pre 4.0	Introduced
6.3	Changed default behavior to create a single file rather than multiple files (see SINGLE_FILE and MULTI_FILE) Changed default behavior to use HOST_ENCODING rather than NETWORK_ENCODING Changed default behavior to use HOST_DECODING rather than NETWORK_DECODING

CDF_DELETE

The CDF_DELETE procedure deletes the specified Common Data Format file. Files deleted include the original `.cdf` file and the `.v0`, `.v1`, etc. files if they exist.

Syntax

CDF_DELETE, *Id*

Arguments

Id

The CDF ID of the file to be deleted, returned from a previous call to CDF_OPEN or CDF_CREATE.

Keywords

None

Examples

```
id = CDF_OPEN('open_close.cdf'); Open a file.
; ... other CDF_ commands ...
CDF_DELETE, id ; Close and Delete the cdf file.
```

Version History

Pre 4.0	Introduced

CDF_DOC

The CDF_DOC procedure retrieves general documentation information about a Common Data Format file.

Syntax

CDF_DOC, *Id*, *Version*, *Release*, *Copyright* [, INCREMENT=*variable*]

Arguments

Id

A CDF ID, returned from a previous call to CDF_OPEN or CDF_CREATE.

Version

A named variable in which the version number of the CDF library that created the CDF is returned.

Release

A named variable in which the release number of the CDF library that created the CDF is returned.

Copyright

A named variable in which the copyright notice of the CDF library that created the CDF is returned.

Keywords

INCREMENT

Set this keyword to a named variable that will contain the increment of the CDF library that created the specified CDF file.

Examples

```
id=CDF_CREATE('VersionCheck')   ; Create a CDF file.
CDF_DOC, id, vers, rel, copy, INCREMENT=incr
PRINT,'File Written Using CDF', vers, rel, incr, $
   FORMAT='(A,I1,".",I1,"r",I2)'
```

```
CDF_CLOSE, id ; Close the CDF file.
```

IDL Output

```
File Written Using CDF2.6
```

Version History

Pre 4.0	Introduced

CDF_ENCODE_EPOCH

The CDF_ENCODE_EPOCH function encodes a CDF_EPOCH variable into a string. Four different string formats are available. The default (EPOCH=0) is the standard CDF format, which may be parsed by the CDF_PARSED_EPOCH function or broken down with the CDF_EPOCH procedure.

Syntax

Result = CDF_ENCODE_EPOCH(*Epoch* [, EPOCH={0 | 1 | 2 | 3}])

Return Value

Returns a string containing the encoded CDF_EPOCH variable.

Arguments

Epoch

The double-precision CDF_EPOCH value to be encoded. For more information about CDF_EPOCH values, see “Data Types” in the *CDF User’s Guide*.

Keywords

EPOCH

Set this keyword equal to one of the following integer values, specifying the epoch mode to use for output of the epoch date string:

Value	Date Format
0	`DD-Mon-YYYY hh:mm:ss.ccc` (This is the default)
1	`YYYYMMDD.ttttttt`
2	`YYYYMMDDhhmmss`
3	`YYYY-MM-DDThh:mm:ss.cccZ` (The characters `T` and `Z` are the CDF_EPOCH type 3 place holders)

where:

Date Element	Represents
`DD`	the day of the month (1-31)
`Mon`	the abbreviated month name: (`Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, or `Dec`)
`MM`	the month number (1-12)
`YYYY`	the year (A.D.)
`hh`	the hour (0-23)
`mm`	the minute (0-59)
`ss`	the second (0-59)
`ccc`	the millisecond (0-999)
`ttttttt`	the fraction of the day (*e.g.* 2500000 is 6 am).

Examples

```
epoch_string = '04-Dec-1995 20:19:18.176'
epoch = CDF_PARSE_EPOCH(epoch_string)
HELP, epoch_string, epoch

; Create encode strings:
encode0 = CDF_ENCODE_EPOCH(epoch, EPOCH=0)
encode1 = CDF_ENCODE_EPOCH(epoch, EPOCH=1)
encode2 = CDF_ENCODE_EPOCH(epoch, EPOCH=2)
encode3 = CDF_ENCODE_EPOCH(epoch, EPOCH=3)

; Compare encoding formats:
HELP, encode0, encode1, encode2, encode3
```

IDL Output

```
EPOCH_STRING    STRING    = '04-Dec-1995 20:19:18.176'
EPOCH           DOUBLE    =    6.2985328e+13

ENCODE0         STRING    = '04-Dec-1995 20:19:18.176'
ENCODE1         STRING    = '19951204.8467381'
ENCODE2         STRING    = '19951204201918'
ENCODE3         STRING    = '1995-12-04T20:19:18.176Z'
```

Version History

4.0.1b	Introduced

See Also

CDF_EPOCH, CDF_PARSE_EPOCH

CDF_ENCODE_EPOCH16

The CDF_ENCODE_EPOCH16 function encodes a CDF_EPOCH16 value into the standard date and time character string.

Syntax

Result = CDF_ENCODE_EPOCH16(*Epoch16* [, EPOCH={0 | 1 | 2 | 3}])

Return Value

Returns the string representation of the given CDF_EPOCH16 value.

Arguments

Epoch16

The double-precision CDF_EPOCH16 value to be encoded.

Note

The CDF_EPOCH16 value can be obtained by calling CDF_EPOCH16 with the COMPUTE keyword, or by calling CDF_PARSE_EPOCH16.

Keywords

EPOCH

Set this keyword equal to one of the following integer values, specifying the epoch mode to use for output of the epoch date string:

Value	Date Format
0	`DD-Mon-YYYY hh:mm:ss.ccc.uuu.nnn.ppp` (This is the default)
1	`YYYYMMDD.ttttttttttttttt`
2	`YYYYMMDDss`
3	`YYYY-MM-DDThh:mm:ss.ccc.uuu.nnn.pppZ` (The characters `T` and `Z` are the CDF_EPOCH16 type 3 place holders)

where:

Date Element	Represents
`DD`	the day of the month (1-31)
`Mon`	the abbreviated month name: (`Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, or `Dec`)
`MM`	the month number (1-12)
`YYYY`	the year (A.D.)
`hh`	the hour (0-23)
`mm`	the minute (0-59)
`ss`	the second (0-59)
`ccc`	the millisecond (0-999)
`uuu`	the microsecond (0-999)
`nnn`	the nanosecond (0-999)
`ppp`	the picosecond (0-999)
`tttttttttttttt`	the fraction of the day (*e.g.* 500000000000000 is noon).

Examples

```
test_string = '04-Dec-2005 20:19:18.176.214.648.000'
test_epoch = CDF_PARSE_EPOCH16(test_string)
PRINT, CDF_ENCODE_EPOCH16(test_epoch)
```

IDL Output

```
04-Dec-2005 20:19:18.176.214.648.000
```

Version History

6.3	Introduced

See Also

CDF_EPOCH16, CDF_PARSE_EPOCH16

CDF_EPOCH

The CDF_EPOCH procedure computes or breaks down CDF_EPOCH values in a CDF file. When computing an epoch, any missing value is considered to be zero.

If you supply a value for the Epoch argument and set the BREAKDOWN_EPOCH keyword, CDF_EPOCH will compute the values of the Year, Month, Day, etc. and insert the values into the named variables you supply. If you specify the Year (and optionally, the Month, Day, etc.) and set the COMPUTE_EPOCH keyword, CDF_EPOCH will compute the epoch and place the value in the named variable supplied as the Epoch parameter.

Note

You must set either the BREAKDOWN_EPOCH or COMPUTE_EPOCH keyword.

Syntax

CDF_EPOCH, *Epoch*, *Year* [, *Month*, *Day*, *Hour*, *Minute*, *Second*, *Milli*]
[, /BREAKDOWN_EPOCH] [, /COMPUTE_EPOCH]

Arguments

Epoch

The Epoch value to be broken down, or a named variable that will contain the computed epoch will be placed. The Epoch value is the number of milliseconds since `01-Jan-0000 00:00:00.000`

Note

“Year zero” is a convention chosen by NSSDC to measure epoch values. This date is more commonly referred to as 1 BC. Remember that 1 BC was a leap year. The Epoch is defined as the number of milliseconds since `01-Jan-0000 00:00:00.000`, as computed using the CDF library’s internal date routines. The CDF date/time calculations do not take into account the changes to the Gregorian calendar, and cannot be directly converted into Julian date/times. To convert between CDF epochs and date/times, use the CDF_EPOCH routine with either the BREAKDOWN_EPOCH or CONVERT_EPOCH keywords.

Year

If COMPUTE_EPOCH is set, a four-digit integer representing the year.

If BREAKDOWN_EPOCH is set, a named variable that will contain the year.

Month

If COMPUTE_EPOCH is set, an integer between 1 and 12 representing the month. Alternately, you can set *Month* equal to zero, in which case the *Day* argument can take on any value between 1-366.

If BREAKDOWN_EPOCH is set, a named variable that will contain the numeric month value.

Day

If COMPUTE_EPOCH is set, an integer between 1 and 31 representing the day of the month. Alternately, if the *Month* argument is set equal to zero, *Day* can be an integer between 1-366 representing the day of the year.

If BREAKDOWN_EPOCH is set, a named variable that will contain the numeric day of the month value.

Hour

If COMPUTE_EPOCH is set, an integer between 0 and 23 representing the hour of the day.

If BREAKDOWN_EPOCH is set, a named variable that will contain the numeric hour of the day value.

Minute

If COMPUTE_EPOCH is set, an integer between 0 and 59 representing the minute of the hour.

If BREAKDOWN_EPOCH is set, a named variable that will contain the numeric minute of the hour value.

Second

If COMPUTE_EPOCH is set, an integer between 0 and 59 representing the second of the minute.

If BREAKDOWN_EPOCH is set, a named variable that will contain the numeric second of the minute value.

Milli

If COMPUTE_EPOCH is set, an integer between 0 and 999 representing the millisecond. Alternately, if the *Hour*, *Minute*, and *Second* arguments are set all equal to zero, *Milli* can be an integer between 0-86400000 representing the millisecond of the day.

If BREAKDOWN_EPOCH is set, a named variable that will contain the numeric millisecond value.

Keywords

BREAKDOWN_EPOCH

Set this keyword to break down the value of the *Epoch* argument into its component parts, storing the resulting year, month, day, *etc*. values in the variables specified by the corresponding arguments.

COMPUTE_EPOCH

Set this keyword to compute the value of *Epoch* from the values specified by the *Year*, *Month*, *Day*, *etc*. arguments.

Examples

To compute the epoch value of September 20, 1992 at 3:00 am:

```
CDF_EPOCH, MergeDate, 1992, 9, 20, 3, /COMPUTE_EPOCH
```

To break down the given epoch value into standard date components:

```
CDF_EPOCH, 4.7107656e13, yr, mo, dy, hr, mn, sc, milli, /BREAK
```

Version History

Pre 4.0	Introduced

CDF_EPOCH16

The CDF_EPOCH16 procedure computes or breaks down a CDF_EPOCH16 value in a CDF file. When computing an epoch, any missing value is considered to be zero.

If you supply a value for the *Epoch* argument and set the BREAKDOWN_EPOCH keyword, CDF_EPOCH16 will compute the values of the *Year*, *Month*, *Day*, *etc*. and insert the values into the named variables you supply.

If you specify the *Year* (and optionally, the *Month*, *Day*, *etc*.) and set the COMPUTE_EPOCH keyword, CDF_EPOCH16 will compute the epoch and place the value in the named variable supplied as the *Epoch* parameter.

Note

You *must* set either the BREAKDOWN_EPOCH or COMPUTE_EPOCH keyword.

Syntax

CDF_EPOCH16, *Epoch*, *Year* [, *Month*, *Day*, *Hour*, *Minute*, *Second*, *Milli*, *Micro*, *Nano*, *Pico*] [, /BREAKDOWN_EPOCH] [, /COMPUTE_EPOCH]

Arguments

Epoch

The Epoch value to be broken down, or a named variable that will contain the computed epoch will be placed. The Epoch value is the number of picoseconds since `01-Jan-0000 00:00:00.000.000.000.000`.

Note

"Year zero" is a convention chosen by CDF to measure epoch values. This date is more commonly referred to as 1 BC. Remember that 1 BC was a leap year. The Epoch is defined as the number of picoseconds since `01-Jan-0000 00:00:00.000.000.000.000`, as computed using the CDF library's internal date routines. The CDF date/time calculations do not take into account the changes to the Gregorian calendar, and cannot be directly converted into Julian date/times. To convert between CDF epochs and date/times, use the CDF_EPOCH16 routine with either the BREAKDOWN_EPOCH or CONVERT_EPOCH keywords.

Year

If COMPUTE_EPOCH is set, a four-digit integer representing the year.

If BREAKDOWN_EPOCH is set, a named variable that will contain the year.

Month

If COMPUTE_EPOCH is set, an integer between 1 and 12 representing the month. Alternately, you can set *Month* equal to zero, in which case the *Day* argument can take on any value between 1-366.

If BREAKDOWN_EPOCH is set, a named variable that will contain the numeric month value.

Day

If COMPUTE_EPOCH is set, an integer between 1 and 31 representing the day of the month. Alternately, if the *Month* argument is set equal to zero, *Day* can be an integer between 1-366 representing the day of the year.

If BREAKDOWN_EPOCH is set, a named variable that will contain the numeric day of the month value.

Hour

If COMPUTE_EPOCH is set, an integer between 0 and 23 representing the hour of the day.

If BREAKDOWN_EPOCH is set, a named variable that will contain the numeric hour of the day value.

Minute

If COMPUTE_EPOCH is set, an integer between 0 and 59 representing the minute of the hour.

If BREAKDOWN_EPOCH is set, a named variable that will contain the numeric minute of the hour value.

Second

If COMPUTE_EPOCH is set, an integer between 0 and 59 representing the second of the minute.

If BREAKDOWN_EPOCH is set, a named variable that will contain the numeric second of the minute value.

Milli

If COMPUTE_EPOCH is set, an integer between 0 and 999 representing the millisecond.

If BREAKDOWN_EPOCH is set, a named variable that will contain the numeric millisecond value.

Micro

If COMPUTE_EPOCH is set, an integer between 0 and 999 representing the microsecond.

If BREAKDOWN_EPOCH is set, a named variable that will contain the numeric microsecond value.

Nano

If COMPUTE_EPOCH is set, an integer between 0 and 999 representing the nanosecond.

If BREAKDOWN_EPOCH is set, a named variable that will contain the numeric nanosecond value.

Pico

If COMPUTE_EPOCH is set, an integer between 0 and 999 representing the picosecond.

If BREAKDOWN_EPOCH is set, a named variable that will contain the numeric picosecond value.

Keywords

BREAKDOWN_EPOCH

Set this keyword to break down the value of the *Epoch* argument into its component parts, storing the resulting year, month, day, *etc*. values in the variables specified by the corresponding arguments.

COMPUTE_EPOCH

Set this keyword to compute the value of *Epoch* from the values specified by the *Year*, *Month*, *Day*, *etc*. arguments.

Examples

To compute the epoch value of September 20, 2005 at 3:05:46:02:156 am:

```
CDF_EPOCH16, epoch, 2005, 9, 20, 3, 5, 46, 27, 2, 156, $
   /COMPUTE_EPOCH
```

To break down the given epoch value into standard date components:

```
CDF_EPOCH16, epoch, yr, mo, dy, hr, min, sec, milli, micro, pico, $
   /BREAKDOWN_EPOCH
```

Version History

6.3	Introduced

CDF_ERROR

The CDF_ERROR function returns a a short explanation of a given status code returned from a Common Data Format file.

Syntax

Result = CDF_ERROR(*Status*)

Return Value

Returns a string containing a status code explanation.

Arguments

Status

The status code to be explained.

Keywords

None

Version History

Pre 4.0	Introduced

CDF_EXISTS

The CDF_EXISTS function returns true if the Common Data Format (CDF) scientific data format library is supported on the current IDL platform.

This routine is written in the IDL language. Its source code can be found in the file `cdf_exists.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = CDF_EXISTS( )

Return Value

Returns a 1 (True) if the library is supported or a 0 (False) if the library is not supported.

Arguments

None

Keywords

None

Examples

The following IDL command prints an error message if the CDF library is not available:

```
IF CDF_EXISTS() EQ 0 THEN PRINT, 'CDF not supported.'
```

Version History

Pre 4.0	Introduced

CDF_INQUIRE

The CDF_INQUIRE function returns global information about the Common Data Format file. The tags of this structure are described below.

Syntax

Result = CDF_INQUIRE(*Id*)

Return Value

This information is returned in a structure of the form:

```
{ NDIMS:0L, DECODING:"", ENCODING:"", MAJORITY:"", MAXREC:0L, $
  NVARS:0L, NZVARS:0L, NATTS:0L, DIM:LONARR(NDIMS) }
```

Explanation of the Structure Tags

The structure returned by this function consists of the following tags:

Tag	Description
NDIMS	The longword integer specifying the number of dimensions in the rVariables in the current CDF.
DECODING	A string describing the decoding type set in the CDF file, such as ‘MAC_DECODING’ or ‘ALPHAVMSD_ENCODING’.
ENCODING	A string describing the type of encoding used in the CDF file, such as ‘NETWORK_ENCODING’ or ‘SUN_ENCODING’.
MAJORITY	A string describing the majority used in the CDF file. The majority will be either row (‘ROW_MAJOR’) or column (‘COL_MAJOR’).
MAXREC	A longword integer specifying the highest record number written in the rVariables in the current CDF. The MAXREC field will contain the value -1 if no rVariables have yet been written to the CDF.
NVARS	A longword integer specifying the number of rVariables (regular variables) in the CDF.

Table 2-1: CDF_INQUIRE Structure Tags

Tag	Description
NZVARS	A longword integer specifying the number of zVariables in the CDF.
NATTS	A longword integer specifying the number of attributes in the CDF. Note that the number returned in this field includes both global and variable attributes. You can use the GET_NUMATTR keyword to the CDF_CONTROL routine to determine the number of each.
DIM	A vector where each element contains the corresponding dimension size for the rVariables in the current CDF. For 0-dimensional CDF's, this argument contains a single element (a zero).

Table 2-1: CDF_INQUIRE Structure Tags (Continued)

Arguments

Id

A CDF ID, returned from a previous call to CDF_OPEN or CDF_CREATE.

Keywords

None

Examples

```
cdfid = CDF_CREATE('CDFinquire', /HP_ENCODING, /MAC_DECODING)
attr1_id = CDF_ATTCREATE(cdfid, 'GLOBAL_ATT', /GLOBAL_SCOPE)
attr2_id = CDF_ATTCREATE(cdfid,'VARIABLE_ATT', /VARIABLE_SCOPE)
CDF_CONTROL, cdfid, GET_NUMATTRS = num_attrs
PRINT, 'This CDF has ', num_attrs(0), $
   'Global attribute(s) and ', num_attrs(1), $
   'Variable attribute(s).', $
   FORMAT='(A,I2,A,I2,A)'

inquire = CDF_INQUIRE(cdfid)
HELP, inquire, /STRUCT
CDF_DELETE, cdfid ; Delete the CDF file.
```

IDL Output

```
This CDF has  1 Global attribute(s) and  1 Variable attribute(s).

** Structure <4003e0c0>, 9 tags, length=48, refs=1:
   NDIMS           LONG                     0
   DECODING        STRING    'MAC_DECODING'
   ENCODING        STRING    'HP_ENCODING'
   MAJORITY        STRING    'ROW_MAJOR'
   MAXREC          LONG                    -1
   NVARS           LONG                     0
   NZVARS          LONG                     0
   NATTS           LONG                     2
   DIM             LONG      Array(1)
```

Version History

Pre 4.0	Introduced

See Also

CDF_CONTROL, CDF_DOC, CDF_LIB_INFO

CDF_LIB_INFO

The CDF_LIB_INFO procedure returns information about the CDF Library being used by this version of IDL. Information about the version of CDF used to create a particular CDF file can be obtained through CDF_DOC.

Syntax

CDF_LIB_INFO [, COPYRIGHT =*variable*] [, INCREMENT=*variable*] [, RELEASE=*variable*] [, SUBINCREMENT=*variable*] [, VERSION=*variable*]

Arguments

None

Keywords

COPYRIGHT

A named variable in which the copyright notice of the CDF library that this version of IDL is using will be returned.

INCREMENT

A named variable in which the incremental number of the CDF library that this version of IDL is using will be returned.

RELEASE

A named variable in which the release number of the CDF library that this version of IDL is using will be returned.

SUBINCREMENT

A named variable in which the sub incremental character of the CDF library that this version of IDL is using will be returned.

VERSION

A named variable in which the version number of the CDF library that this version of IDL is using will be returned.

Examples

```
CDF_LIB_INFO, VERSION=V, RELEASE=R, COPYRIGHT=C, $
   INCREMENT=I
PRINT, 'IDL ', !version.release, 'uses CDF Library ', $
   V, R, I, FORMAT='(A,A,A,I0,".",I0,".",I0,A)'
PRINT, C
```

IDL Output

```
IDL 6.3 uses CDF Library 3.1.1
Common Data Format (CDF)
(C) Copyright 1990-2005 NASA/GSFC
Space Physics Data Facility
NASA/Goddard Space Flight Center
Greenbelt, Maryland 20771 USA
(Internet -- CDFSUPPORT@LISTSERV.GSFC.NASA.GOV)
```

Version History

Pre 4.0	Introduced

See Also

CDF_DOC

CDF_OPEN

The CDF_OPEN function opens an existing Common Data Format file.

Syntax

Result = CDF_OPEN(*Filename*)

Return Value

If successful, the CDF ID for the file is returned.

Arguments

Filename

A scalar string containing the name of the file to be created. Note that if the desired filename has a `.cdf` ending, you can omit the extension and specify just the first part of the filename. For example, specifying “mydata” would open the file `mydata.cdf`.

Keywords

None

Examples

```
id = CDF_OPEN('open_close.cdf')   ; Open a file.
; ... other CDF_ commands ...
CDF_CLOSE, id ; Close the cdf file.
```

Version History

Pre 4.0	Introduced

CDF_PARSE_EPOCH

The CDF_PARSE_EPOCH function parses a properly-formatted input string into a double-precision value properly formatted for use as a CDF_EPOCH variable.

Note

CDF_EPOCH variables may be unparsed into a variety of formats using the CDF_ENCODE_EPOCH function.

Syntax

Result = CDF_PARSE_EPOCH(*Epoch_string*)

Return Value

Returns the double-precision value of the input string properly formatted for use as a CDF_EPOCH variable.

Arguments

Epoch_string

A formatted string that will be parsed into a double precision value suitable to be used as a CDF_EPOCH value. The format of the date string is:

```
DD-Mon-YYYY hh:mm:ss.ccc
```

where:

Date Element	Represents
`DD`	the day of the month (1-31)
`Mon`	the abbreviated month name: (`Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, or `Dec`)
`YYYY`	the year (A.D.)
`hh`	the hour (0-23)
`mm`	the minute (0-59)

Date Element	Represents
`ss`	the second (0-59)
`ccc`	the millisecond (0-999)

For more information about CDF_EPOCH values, see “Data Types” in the *CDF User’s Guide*.

Keywords

None

Examples

```
test_string = '04-Dec-1995 20:19:18.176'
test_epoch = CDF_PARSE_EPOCH(test_string)
HELP, test_string, test_epoch
PRINT, CDF_ENCODE_EPOCH(test_epoch, EPOCH=0)
```

IDL Output

```
TEST_STRING     STRING    = '04-Dec-1995 20:19:18.176'
TEST_EPOCH      DOUBLE    =    6.2985328e+13

04-Dec-1995 20:19:18.176
```

Version History

4.0.1b	Introduced

See Also

CDF_ENCODE_EPOCH, CDF_EPOCH

CDF_PARSE_EPOCH16

The CDF_PARSE_EPOCH16 function parses a properly-formatted input string into a double-complex value properly formatted for use as a CDF_EPOCH16 variable.

Note

CDF_EPOCH16 variables may be unparsed into a variety of formats using the CDF_ENCODE_EPOCH16 or CDF_EPOCH16 functions.

Syntax

Result = CDF_PARSE_EPOCH16(*Epoch_string*)

Return Value

Returns the double-precision complex value of the input string properly formatted for use as a CDF_EPOCH16 variable.

Arguments

Epoch_string

A formatted string that will be parsed into a double precision complex value suitable to be used as a CDF_EPOCH value. The format of the date string is:

```
DD-Mon-YYYY hh:mm:ss.ccc.uuu.nnn.ppp
```

where:

Date Element	Represents
`DD`	the day of the month (1-31)
`Mon`	the abbreviated month name: (`Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, or `Dec`)
`YYYY`	the year (A.D.)
`hh`	the hour (0-23)
`mm`	the minute (0-59)
`ss`	the second (0-59)

Date Element	Represents
ccc	the millisecond (0-999)
uuu	the microsecond (0-999)
nnn	the nanosecond (0-999)
ppp	the picosecond (0-999)

Keywords

None

Example

```
test_string = '04-Dec-2005 20:19:18.176.214.648.000'
test_epoch = CDF_PARSE_EPOCH16(test_string)
CDF_EPOCH16, test_epoch, year, month, day, hour, min, sec, $
   milli, micro, nano, pico, /BREAKDOWN_EPOCH
HELP, test_string, test_epoch
PRINT, CDF_ENCODE_EPOCH16(test_epoch)
PRINT, year, month, day, hour, min, sec, milli, micro, nano, pico
```

IDL Prints:

```
TEST_STRING     STRING    = '04-Dec-2005 20:19:18.176.214.648.000'
TEST_EPOCH      DCOMPLEX  = (   6.3300947e+10,   1.7621465e+11)

04-Dec-2005 20:19:18.176.214.648.000
2005  12  4  20  19  18  176  214  648  0
```

Version History

6.3	Introduced

See Also

CDF_ENCODE_EPOCH16, CDF_EPOCH16

CDF_SET_CDF27_BACKWARD_COMPATIBLE

The CDF_SET_CDF27_BACKWARD_COMPATIBLE procedure allows users of IDL version 6.3 and later to create a CDF file that can be read by IDL 6.2 or earlier, or by CDF version 2.7.2 or earlier. By default, a CDF file created by IDL 6.3 or later cannot be read by earlier versions.

This procedure must be called prior to calling the CDF_CREATE function. It is useful if you need to create and share CDF files with colleagues who access CDF files using IDL version 6.2 or earlier, or using the CDF library version 2.7.2 or earlier.

In CDF version 2.7.2 and earlier, the maximum CDF file size was 2 Gbytes. This limitation was lifted in CDF version 3.0 with the use of a 64-bit file offset. As a result, users who use a CDF library older than CDF version 3.0 cannot read CDF files that were produced by CDF version 3.0 or a later release. CDF versions 3.0 and later *can* read files that were generated with any of the previous CDF releases.

Note

After calling this routine, the “backward compatible” setting specified persists either until this routine is called again or until the end of the IDL session.

Syntax

CDF_SET_CDF27_BACKWARD_COMPATIBLE [, /YES | /NO]

Arguments

None

Keywords

YES

Set this keyword to create a CDF file that can be read by IDL version 6.2 or earlier, or by CDF version 2.7.2 or earlier. If this keyword is set, the maximum file size is 2 Gbytes.

NO

Set this keyword to create a file that can only be read using IDL version 6.3 or later or CDF version 3.0 or later. This is the default when a CDF file is created.

Examples

Use the following command to create a CDF file that can be read by IDL 6.2 or earlier IDL versions.

```
CDF_SET_CDF27_BACKWARD_COMPATIBLE, /YES
id = CDF_CREATE('myfile.cdf')
CDF_CLOSE, id
```

Version History

6.3	Introduced

CDF_VARCREATE

The CDF_VARCREATE function creates a new variable in a Common Data Format file.

In CDF, *variable* is a generic name for an object that represents data. Data can be scalar (0-dimensional) or multi-dimensional (up to 10-dimensional). Data does not have any associated scientific context; it could represent an independent variable, a dependent variable, a time and date value, an image, the name of an XML file, *etc*. You can describe a variable's relationship to other variables via CDF's *attributes*.

CDF supports two types of variables: zVariables and rVariables. Different zVariables in a CDF data set can have different numbers of dimensions and different dimension sizes. All rVariables in a CDF data set must have the same number of dimensions and the same dimension sizes; this is much less efficient than the zVariable storage mechanism. (rVariables were included in the original version of CDF, and zVariables were added in a later version to address the rVariables' inefficient use of disk space.)

If you are working with a data set created using an early version of CDF, you may need to use rVariables. If you are creating a new CDF data set, you should use zVariables.

Syntax

Result = CDF_VARCREATE(*Id*, *Name* [, *DimVary*] [, /*VariableType*]
[, ALLOCATERECS=*records*] [, DIMENSIONS=*array*]
[, NUMELEM=*characters*] [, /REC_NOVARY | , /REC_VARY]
[, /ZVARIABLE])

Return Value

Returns the variable of the type specified by the chosen keyword.

Arguments

Id

The CDF ID, returned from a previous call to CDF_OPEN or CDF_CREATE.

Name

A string containing the name of the variable to be created.

DimVary

A one-dimensional array containing one element per CDF dimension. If the element is non-zero or the string ‘VARY’, the variable will have variance in that dimension. If the element is zero or the string ‘NOVARY’ then the variable will have no variance with that dimension. If the variable is zero-dimensional, this argument may be omitted.

Keywords

VariableType

You must specify the type variable being created. This is done by setting one of the following keywords:

CDF_BYTE

CDF_CHAR

CDF_DOUBLE

CDF_EPOCH

CDF_EPOCH16

CDF_FLOAT

CDF_INT1

CDF_INT2

CDF_INT4

CDF_REAL4

CDF_REAL8

CDF_UCHAR

CDF_UINT1

CDF_UINT2

CDF_UINT4

If no type is specified, CDF_FLOAT is assumed.

ALLOCATERECS

Set this keyword equal to the desired number of pre-allocated records for this variable in a SINGLE_FILE CDF file. Pre-allocating records ensures that variable data is stored contiguously in the CDF file. For discussion about allocating records, see “Records” in the *CDF User’s Guide*.

DIMENSIONS

Set this keyword to create a new zVariable with the specified dimensions. If this keyword is not set, the variable is assumed to be a scalar. For example:

```
id = CDF_CREATE("cdffile.cdf", [100] )
zid = CDF_VARCREATE(id, "Zvar", [1,1,1], DIM=[10,20,30])
```

Note

Variables created with the DIMENSIONS keyword set are always zVariables.

NUMELEM

The number of elements of the data type at each variable value. This keyword only has meaning for string data types (CDF_CHAR, CDF_UCHAR). This is the number of characters in the string. The default is 1.

REC_NOVARY

If this keyword is set, all records will contain the same information.

REC_VARY

If this keyword is set, all records will contain unique data. This is the default.

ZVARIABLE

Set this keyword to create a zVariable. For example:

```
id = CDF_CREATE("cdffile.cdf", [100] )
zid = CDF_VARCREATE(id, "Zvar", /ZVARIABLE)
```

Note

zVariables are much more efficient than rVariables in their use of disk space. Unless the recipient of your data set is using a very early version of CDF, you should always create zVariables.

Note

Variables created with the DIMENSIONS keyword set are always zVariables.

Examples

In this example, we create a variable, setting the data type from a string variable, which could have been returned by the DATATYPE keyword to a CDF_VARINQ call:

```
; create a file in IDL’s current directory
id = CDF_CREATE('temp.cdf')

VARTYPE = 'CDF_FLOAT'

; Use the _EXTRA keyword and the CREATE_STRUCT function to
; make the appropriate keyword.

VarId = CDF_VARCREATE(Id, 'Pressure', [1,1], $
   NUMELEM=2, _EXTRA=CREATE_STRUCT(VARTYPE,1))
CDF_CLOSE, id ; Close the CDF file.
```

Version History

Pre 4.0	Introduced
6.3	Add support for the CDF_EPOCH16 variable type

CDF_VARDELETE

The CDF_VARDELETE procedure deletes a variable from a SINGLE_FILE CDF file. Note that the variable's entries are also deleted, and that the variables that numerically follow the deleted variable within the CDF file are automatically renumbered. CDF rVariables and zVariables are counted separately within CDF files. Attempting to delete a variable from a MULTI_FILE format CDF file will result in a warning message.

Syntax

CDF_VARDELETE, *Id*, *Variable* [, /ZVARIABLE]

Arguments

Id

The CDF ID of the file containing the Variable to be deleted, returned from a previous call to CDF_OPEN or CDF_CREATE.

Variable

A string containing the name of the variable to be deleted OR the variable number to be deleted. Variable numbers are 0-based in IDL. zVariables and rVariables are counted separately in CDF files.

Keywords

ZVARIABLE

Set this keyword if the Variable is a zVariable and was passed by number. The default is to assume that Variable is an rVariable.

Examples

```
cid = CDF_CREATE('DEMOvardelete',/SINGLE_FILE)

; Create 3 zVariables and 1 rVariable:
var1_id = CDF_VARCREATE(cid, 'rVAR1', /CDF_FLOAT)
var2_id = CDF_VARCREATE(cid, 'zVAR1', /CDF_INT4, /REC_NOVARY, $
   /ZVARIABLE)
var3_id = CDF_VARCREATE(cid, 'zVAR2', /CDF_CHAR, [2,10], $
   NUMELEM=10, DIM=[5,5])
```

```
var4_id = CDF_VARCREATE(cid, 'zVAR3' ,/CDF_REAL8, /ZVARIABLE)

; Check the number of variables:
info = CDF_INQUIRE(cid)
HELP, info.nzvars, info.nvars

; Delete the first and third zvariables:
CDF_VARDELETE, cid, 'zVAR1', /ZVARIABLE
CDF_VARDELETE, cid, 1, /ZVARIABLE

; CAUTION: Remember the variable numbers are zero-based
; and are automatically renumbered.

info = CDF_INQUIRE(cid)
HELP, info.nzvars, info.nvars
varinfo = CDF_VARINQ(cid, 0, /ZVARIABLE)
; check on zVAR2
HELP, varinfo, /STRUCTURE

CDF_DELETE, cid
```

IDL Output

```
<Expression>    LONG      =            3
<Expression>    LONG      =            1

<Expression>    LONG      =            1
<Expression>    LONG      =            1

** Structure <400a3b40>, 8 tags, length=48, refs=1:
   IS_ZVAR         INT              1
   NAME            STRING    'zVAR2'
   DATATYPE        STRING    'CDF_CHAR'
   NUMELEM         LONG                10
   RECVAR          STRING    'VARY'
   DIMVAR          BYTE      Array(2)
   ALLOCATERECS    LONG      Array(2)
   DIM             LONG      Array(1)
```

Version History

4.0.1b	Introduced

See Also

CDF_ATTDELETE, CDF_CONTROL, CDF_VARCREATE, CDF_VARINQ

CDF_VARGET

The CDF_VARGET procedure reads multiple values from a Common Data Format file variable. By default, all elements of a record are read. If INTERVAL and/or OFFSET are specified but no COUNT is specified, CDF_VARGET attempts to get as many elements of each record as possible.

Syntax

CDF_VARGET, *Id*, *Variable*, *Value* [, COUNT=*vector*] [, INTERVAL=*vector*] [, OFFSET=*vector*] [, REC_COUNT=*records*] [, REC_INTERVAL=*value*] [, REC_START=*record*] [, /STRING{data in CDF file must be type CDF_CHAR or CDF_UCHAR}] [, /ZVARIABLE]

Arguments

Id

The CDF ID, returned from a previous call to CDF_OPEN or CDF_CREATE

Variable

A string containing the name of the variable *or* the variable number being read.

Value

A named variable in which the values of the variable are returned.

Keywords

COUNT

An optional vector containing the counts to be used in reading Value. The default is to read all elements in each record, taking into account INTERVAL and OFFSET.

INTERVAL

A vector specifying the interval between values in each dimension. The default value is 1 for each dimension.

OFFSET

A vector specifying the array indices within the specified record(s) at which to begin writing. OFFSET is a 1-dimensional array containing one element per CDF dimension. The default value is zero for each dimension.

REC_COUNT

The number of records to read. The default is 1.

REC_INTERVAL

The interval between records when reading multiple records. The default value is 1.

REC_START

The record number at which to start reading. The default is 0.

STRING

Set this keyword to return CDF_CHAR and CDF_UCHAR data from the CDF file into *Value* as string data rather than byte data. This keyword is ignored if the data in the CDF file is not of type CDF_CHAR or CDF_UCHAR.

ZVARIABLE

If *Variable* is a variable ID (as opposed to a variable name) and the variable is a zVariable, set this flag to indicate that the variable ID is a zVariable ID. The default is to assume that *Variable* is an rVariable ID.

Examples

```
; Create a CDF file,  and make a few variables:
id = CDF_CREATE('DEMOvargets')
vid1 = CDF_VARCREATE(id, 'VAR1', /CDF_CHAR, NUMELEM=15)
vid2=CDF_VARCREATE(id, 'VAR2', /CDF_UCHAR, NUMELEM=10)
CDF_VARPUT, id, vid1, BINDGEN(15, 2)+55, COUNT=2
CDF_VARPUT, id, vid2, ['IDLandCDF ', 'AreWayCool'

; Retrieve the CDF_CHAR array as byte data:
CDF_VARGET, id,'VAR1',var1_byte,REC_COUNT=2
HELP, var1_byte

;Retrieve the CDF_CHAR array as string data:
CDF_VARGET, id, 'VAR1', var1_string, REC_COUNT=2, /STRING
HELP, var1_string
```

```
; For demonstration purposes, use the 'VAR2' variable number to
; access 'VAR2' for the duration of this example:

var2num = CDF_VARNUM(id, 'VAR2')
HELP, var2num

; Rename 'VAR2' to 'VAR_STRING_2':
CDF_VARRENAME, id, var2num, 'VAR_STRING_2'

; Examine VAR_STRING_2 with CDF_VARINQ:
VAR2_INQ = CDF_VARINQ(id, var2num)
HELP, VAR2_INQ, /STRUCTURE

; Read in and print out VAR_STRING_2:
CDF_VARGET, id, var2num, var2_string, /STRING, REC_COUNT=2
PRINT, var2_string

CDF_DELETE, id ; Delete the CDF file
```

IDL Output

```
% CDF_VARGET: Warning: converting data to unsigned bytes
```

This warning message indicates that the data was stored in the CDF file with type CDF_CHAR (signed 1-byte characters), but was retrieved by IDL with type BYTE (unsigned byte). To turn this warning message off, set !QUIET=1.

```
VAR1_BYTE       BYTE      = Array(15,  2)

VAR1_STRING     STRING    = Array(2)

VAR2NUM         LONG      =            1

** Structure <400b1600>, 6 tags, length=33, refs=1:
   IS_ZVAR         INT                    0
   NAME            STRING     'VAR_STRING_2'
   DATATYPE        STRING        'CDF_UCHAR'
   NUMELEM         LONG                    10
   RECVAR          STRING             'VARY'
   DIMVAR          BYTE                     0

IDLandCDF  AreWayCool
```

Version History

Pre 4.0	Introduced

CDF_VARGET1

The CDF_VARGET1 procedure reads one value from a CDF file variable.

Syntax

CDF_VARGET1, *Id*, *Variable*, *Value* [, OFFSET=*vector*] [, REC_START=*record*]
[, /STRING{data in CDF file must be type CDF_CHAR or CDF_UCHAR}]
[, /ZVARIABLE]

Arguments

Id

The CDF ID, returned from a previous call to CDF_OPEN or CDF_CREATE.

Variable

A string containing the name or an integer containing the index of the variable being inquired.

Value

A named variable in which the value of the variable is returned.

Keywords

OFFSET

A vector specifying the array indices within the specified record(s) at which to begin reading. OFFSET is a 1-dimensional array containing one element per CDF dimension. The default value is 0 for each dimension.

REC_START

The record number at which to start reading. The default is 0.

STRING

Set this keyword to return CDF_CHAR and CDF_UCHAR data from the CDF file into Value as string data rather than byte data. This keyword is ignored if the data in the CDF file is not of type CDF_CHAR or CDF_UCHAR.

ZVARIABLE

If Variable is a variable ID (as opposed to a variable name) and the variable is a zVariable, set this flag to indicate that the variable ID is a zVariable ID. The default is to assume that Variable is an rVariable ID.

Examples

See the example for "CDF_VARCREATE" on page 96.

Version History

Pre 4.0	Introduced

CDF_VARINQ

The CDF_VARINQ function returns a structure containing information about the specified variable in a Common Data Format file.

Syntax

Result = CDF_VARINQ(*Id*, *Variable* [, /ZVARIABLE])

Return Value

The returned structure has the form:

```
{ IS_ZVAR:0, NAME:"", DATATYPE:"", NUMELEM:0L, $
   RECVAR:"", DIMVAR:BYTARR(...) [, DIM:LONARR(...)]}
```

Note

The DIM field is included in the structure only if IS_ZVAR is one.

Explanation of the Structure Tags

The following table provides structure tag information.

Tag	Description
IS_ZVAR	This field will contain a 1 if the variable is a zVariable or a 0 if it is an rVariable.
NAME	The name of the variable.
DATATYPE	A string describing the data type of the variable. The string has the form 'CDF_XXX' where XXX is FLOAT, DOUBLE, EPOCH, UCHAR, etc.
NUMELEM	The number of elements of the data type at each variable value. This is always 1 except in the case of string type variables (CDF_CHAR, CDF_UCHAR).
RECVAR	A string describing the record variance of the variable. This is either the string 'VARY' or 'NOVARY'.

Table 2-2: CDF_VARINQ Structure Tags

Tag	Description
DIMVAR	An array of bytes. The value of each element is zero if there is no variance with that dimension and one if there is variance. For zero-dimensional CDFs, DIMVAR will have one element whose value is zero.
DIM	An array of longs. The value of each element corresponds to the dimension of the variable. This field is only included in the structure if the variable is a zVariable.

Table 2-2: CDF_VARINQ Structure Tags (Continued)

Arguments

Id

The CDF ID, returned from a previous call to CDF_OPEN or CDF_CREATE.

Variable

A string containing the name or an integer containing the index of the variable being inquired.

Keywords

ZVARIABLE

If Variable is a variable ID (as opposed to a variable name) and the variable is a zVariable, set this flag to indicate that the variable ID is a zVariable ID. The default is to assume that Variable is an rVariable ID.

Examples

See the example for “CDF_VARGET” on page 103.

Version History

Pre 4.0	Introduced

CDF_VARNUM

The CDF_VARNUM function returns the variable number associated with a given variable name in a Common Data Format file.

Syntax

Result = CDF_VARNUM(*Id*, *VarName* [, *IsZVar*])

Return Value

Returns the variable number of a specified variable name. If the specified variable cannot be found in the CDF file, CDF_VARNUM returns the scalar -1.

Arguments

Id

The CDF ID, returned from a previous call to CDF_OPEN or CDF_CREATE.

VarName

A string containing the name of the variable.

IsZVar

A named variable into which CDF_VARNUM will place a 1 to indicate that the referenced variable is a zVariable or a 0 to indicate that it is an rVariable.

Keywords

None

Examples

See the example for "CDF_VARGET" on page 103.

Version History

Pre 4.0	Introduced

CDF_VARPUT

The CDF_VARPUT procedure writes a value to a variable in a Common Data Format file. This function provides equivalent functionality to the C routines CDFvarPut and CDFvarHyperPut.

Syntax

CDF_VARPUT, *Id*, *Variable*, *Value* [, COUNT=*vector*] [, INTERVAL=*vector*]
[, OFFSET=*vector*] [, REC_INTERVAL=*value*] [, REC_START=*record*]
[, /ZVARIABLE]

Arguments

Id

The CDF ID, returned from a previous call to CDF_OPEN or CDF_CREATE.

Variable

A string containing the name or number of the variable being written.

Value

The value to write. If the value has 1 more dimension than the CDF, multiple records will be written.

Keywords

COUNT

An optional vector containing the counts to be used in writing Value. Note that counts do not have to match the dimensions of Value. The default count is to use the dimensions of *Value*.

INTERVAL

A vector specifying the interval between values in each dimension. The default value is 1 in each dimension.

OFFSET

A vector specifying the array indices within the specified record(s) at which to begin writing. OFFSET is a 1-dimensional array containing one element per CDF dimension. The default value is zero in each dimension.

REC_INTERVAL

The interval between records being written when writing multiple records. The default value is 1.

REC_START

The record number at which to start writing. The default is 0.

ZVARIABLE

If *Variable* is a variable ID (as opposed to a variable name) and the variable is a zVariable, set this flag to indicate that the variable ID is a zVariable ID. The default is to assume that *Variable* is an rVariable ID.

Examples

```
id= CDF_CREATE('mycdf', [5,10], /NETWORK_ENCODING, /ROW_MAJOR)
varid= CDF_VARCREATE(id, 'V1', [1,1], /CDF_FLOAT, /REC_VARY)
```

To write the value 42.0 into record 12, third row, fourth column:

```
CDF_VARPUT, id, varid, 42, REC_START=12, OFFSET=[2,3]
```

To write 3 records, skipping every other record, starting at record 2, writing every other entry of each record. Note that in this example we write 25 values into each record written:

```
CDF_VARPUT, id, varid, FINDGEN(5,5,3), INTERVAL=[2,1], $
   REC_INTERVAL=2, REC_START=2
CDF_DELETE, id
```

Version History

Pre 4.0	Introduced

CDF_VARRENAME

The CDF_VARRENAME procedure renames an existing variable in a Common Data Format file.

Syntax

CDF_VARRENAME, *Id*, *OldVariable*, *NewName* [, /ZVARIABLE]

Arguments

Id

The CDF ID, returned from a previous call to CDF_OPEN or CDF_CREATE.

OldVariable

A string containing the current name of the variable *or* the variable number to be renamed.

NewName

A string containing the new name for the variable.

Keywords

ZVARIABLE

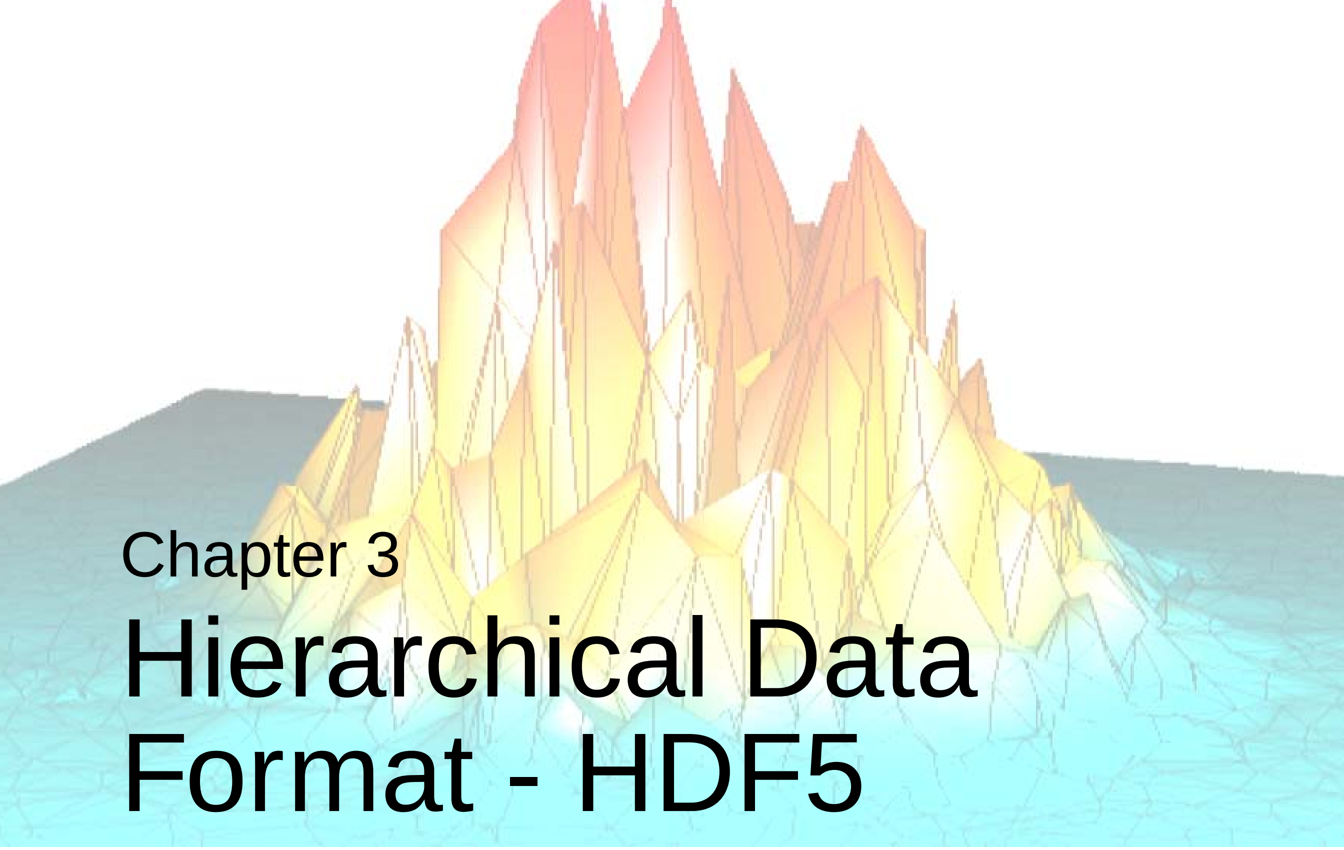
If OldVariable is a variable ID (as opposed to a variable name) and the variable is a zVariable, set this flag to indicate that the variable ID is a zVariable ID. The default is to assume that OldVariable is an rVariable ID.

Examples

See the example for "CDF_VARGET" on page 103.

Version History

Pre 4.0	Introduced



Chapter 3

Hierarchical Data Format - HDF5

This chapter details the interface routines for the Hierarchical Data Format version 5. The following topics are covered in this chapter:

Overview of the HDF Version 5 Format

The Hierarchical Data Format version 5 file format is designed for scientific data consisting of a hierarchy of datasets and attributes (or metadata). HDF is a product of the National Center for Supercomputing Applications (NCSA), which supplies the underlying C-language library; IDL provides access to this library via a set of procedures and functions contained in a dynamically loadable module (DLM).

This version of IDL supports HDF5 5-1.6.3. IDL's HDF5 routines all begin with the prefix "H5_" or "H5*_".

For more information on HDF5 see:

`http://hdf.ncsa.uiuc.edu/HDF5/`

The HDF5 Format

Hierarchical Data Format files are organized in a hierarchical structure. The two primary structures are:

- The HDF5 group — a grouping structure containing instances of zero or more groups or datasets, together with supporting metadata.
- The HDF5 dataset — a multidimensional array of data elements, together with supporting metadata.

HDF attributes are small named datasets that are attached to primary datasets, groups, or named datatypes.

HDF4 versus HDF5

HDF5 was designed to address some of the limitations of the HDF4 format, in addition to providing new functionality.

The limitations of the HDF4 format included:

- A file cannot store more than 20,000 complex objects and cannot be larger than 2 gigabytes;
- The data models are inconsistent, there are too many object types, and datatypes are too restrictive;
- The C library source was old and complex, did not support parallel I/O effectively, and was not threadsafe.

The new HDF5 includes the following improvements:

- Larger files may be stored and more objects per file may be included.
- A more comprehensive data model with two basic structures: multidimensional datasets and groups.
- Simpler, better-engineered library and API, with support for parallel I/O and threads.

Note

The HDF5 format is not compatible with HDF4, although a conversion routine (`h4toh5`) is available from NCSA (`http://hdf.ncsa.uiuc.edu/h4toh5/`).

The IDL HDF5 Library

The IDL HDF5 library consists of an almost direct mapping between the HDF5 library functions and the IDL functions and procedures. The relationship between the IDL routines and the HDF5 library is described in the following subsections.

Routine Names

The IDL routine names are typically identical to the HDF5 function names, with the exception that an underscore is added between the prefix and the actual function. For example, the C function `H5get_libversion()` is implemented by the `IDL` function `H5_GET_LIBVERSION`.

The IDL HDF5 library contains the following function categories:

Prefix	Category	Purpose
H5	Library	General library tasks
H5A	Attribute	Manipulate attribute datasets
H5D	Dataset	Manipulate general datasets
H5F	File	Create, open, and close files
H5G	Group	Handle groups of other groups or datasets
H5I	Identifier	Query object identifiers
H5R	Reference	Reference identifiers
H5S	Dataspace	Handle dataspace dimensions and selection
H5T	Datatype	Handle dataset element information

Table 3-1: HDF Function Categories

Functions Versus Procedures

HDF5 functions that only return an error code are typically implemented as IDL procedures. An example is `H5F_CLOSE`, which takes a single file identifier number as the argument and closes the file. HDF5 functions that return values are implemented as IDL functions. An example is `H5F_OPEN`, which takes a filename as the argument and returns a file identifier number.

Error Handling

All HDF5 functions that return an error or status code are checked for failure. If an error occurs, the HDF5 error handling code is called to retrieve the internal HDF5 error message. This error message is printed to the output window, and program execution stops.

Dimension Order

HDF5 uses C row-major ordering instead of IDL column-major ordering. For row major, the first listed dimension varies slowest, while for column major the first listed dimension varies fastest. The IDL HDF5 library handles this difference by automatically reversing the dimensions for all functions that accept lists of dimensions.

Note

Only the order in which the dimensions are listed is affected; in both the HDF5 file and in IDL memory, the layout of the data is identical.

For example, an HDF5 file may be known to contain a dataset with dimensions [5][10][50], either as declared in the C code, or from the output from the `h5dump` utility. When this dataset is read into IDL, the array will have the dimensions listed as [50, 10, 5], using the output from the IDL help function.

Note

In both the C program used to create the file and in IDL memory after reading the dataset, the values with dimension 50 will be contiguous.

IDL HDF5 Limitations

The IDL HDF5 library currently has the following limitations:

- Conversion cannot be forced from the HDF5 datatype to a different IDL type (such as single-precision instead of double), although data can be converted after reading from the file.
- Only the first (or top) error message from the HDF5 error stack is printed.
- Variable-length and opaque datatypes are currently ignored.
- The low-level property interface (H5P) is not exposed.

HDF5 Datatypes

In HDF5, a *datatype* is an object that describes the storage format of the individual data points of a data set. There are two categories of datatypes; *atomic* and *compound* datatypes:

- *Atomic datatypes* cannot be decomposed into smaller units at the API level.
- *Compound datatypes* are a collection of one or more atomic types or small arrays of such types. Compound datatypes are similar to a struct in C or a common block in Fortran. See "Compound Datatypes" on page 124 for additional details.

In addition, HDF5 uses the following terms for different datatype concepts:

- A *named datatype* is a datatype that is named and stored in a file. Naming is permanent; a datatype cannot be changed after being named. Named datatypes are created from in-memory datatypes using the H5T_COMMIT routine.
- An *opaque datatype* is a mechanism for describing data which cannot be otherwise described by HDF5. The only properties associated with opaque types are the size in bytes and an ASCII tag string. See "Opaque Datatypes" on page 125 for additional details.
- An *enumeration datatype* is a one-to-one mapping between a set of symbols and an ordered set of integer values. The symbols are passed between IDL and the underlying HDF5 library as character strings. All the values for a particular enumeration datatype are of the same integer type. See "Enumeration Datatypes" on page 127 for additional details.
- A *variable length array* datatype is a sequence of existing datatypes (atomic, variable length, or compound) which are not fixed in length from one dataset location to another. See "Variable Length Array Datatypes" on page 129 for additional details.

Compound Datatypes

HDF5 compound datatypes can be compared to C structures, Fortran structures, or SQL records. Compound datatypes can be nested; there is no limitation to the complexity of a compound datatype. Each member of a compound datatype must have a descriptive name, which is the key used to uniquely identify the member within the compound datatype.

Use one of the H5T_COMPOUND_CREATE or H5T_IDL_CREATE routines to create compound datatypes. Use the following routines to work with compound datatypes:

- H5T_GET_MEMBER_CLASS
- H5T_GET_MEMBER_INDEX
- H5T_GET_MEMBER_NAME
- H5T_GET_MEMBER_OFFSET
- H5T_GET_MEMBER_TYPE
- H5T_GET_MEMBER_VALUE
- H5T_GET_NMEMBERS

Example

See H5F_CREATE for an extensive example using compound datatypes.

Opaque Datatypes

An opaque datatype contains a series of bytes. It always contains a single element, regardless of the length of the series of bytes it contains.

When an IDL variable is written to a dataset or attribute defined as an opaque datatype, it is written as a string of bytes with no demarcation. When data in a opaque datatype is read into an IDL variable, it is returned as byte array. Use the FIX routine to convert the returned byte array to the appropriate IDL data type.

Use the H5T_IDL_CREATE routine with the OPAQUE keyword to create opaque datatypes. To create an opaque array, use an opaque datatype with the H5T_ARRAY_CREATE routine. A single string tag can be assigned to an opaque datatype to provide auxiliary information about what is contained therein. Create tags using the H5T_SET_TAG routine; retrieve tags using the H5T_GET_TAG routine. HDF5 limits the length of the tag to 255 characters.

Example

The following example creates an opaque datatype and stores within it a 20-element integer array.

```
; Create a file to hold the data
file = 'h5_test.h5'
fid = H5F_CREATE(file)

; Create some data
data = INDGEN(20)

; Create an opaque datatype
dt = H5T_IDL_CREATE(data, /OPAQUE)

; Create a single element dataspace
ds = H5S_CREATE_SIMPLE(1)

; Create and write the dataset
d = H5D_CREATE(fid, 'dataset', dt, ds)
H5D_WRITE, d, data

; Close the file
H5F_CLOSE, fid

; Reopen file for reading
fid = H5F_OPEN(file)

; Read in the data
d = H5D_OPEN(fid, 'dataset')
```

```
result = H5D_READ(d)

; Close the file
H5F_CLOSE, fid

HELP, result
```

IDL prints:

```
RESULT          BYTE      = Array[40]
```

Note that the result is a 40-element byte array, since each integer requires two bytes.

Enumeration Datatypes

An enumeration datatype consists of a set of (*Name*, *Value*) pairs, where:

- *Name* is a scalar string that is unique within the datatype (a given name string can only be associated with a single value)
- *Value* is a scalar integer that is unique within the datatype

Note

Name/value pairs must be assigned to the datatype *before* it is used to create a dataset. The dataset stores the state of the datatype at the time the dataset is created; additional changes to the datatype will not be reflected in the dataset.

Create the enumeration datatype using the H5T_ENUM_CREATE function. Once you have created an enumeration datatype:

- use the H5T_ENUM_INSERT procedure to associate a single name/value pair with the datatype
- use the H5T_ENUM_VALUEOF function to retrieve the value associated with a single name
- use the H5T_ENUM_NAMEOF function to retrieve the name associated with a single value

These routines replicate the facilities provided by the underlying HDF5 library, which deals only with single name/value pairs. To make it easier to read and write entire enumerated lists, IDL provides two helper routines at package the name/value pairs in arrays of IDL IDL_H5_ENUM structures, which have the following definition:

```
{IDL_H5_ENUM, NAME:'', VALUE:0}
```

The routines are:

- H5T_ENUM_SET_DATA associates multiple name/value pairs with an enumeration datatype in a single operation. Data can be provided either as a string array of names and an integer array of values or as a single array of IDL_H5_ENUM structures.
- H5T_ENUM_GET_DATA retrieves multiple name/value pairs from an enumeration datatype in a single operation. Data are returned in an array of IDL_H5_ENUM structures.

The H5T_ENUM_VALUES_TO_NAMES function is a helper routine that lets you retrieve the names associated with an array of values in a single operation.

The following routines may also be useful when working with enumeration datatypes:

H5T_GET_MEMBER_INDEX, H5T_GET_MEMBER_NAME, H5T_GET_MEMBER_VALUE

Example

The following example creates an enumeration datatype and saves it to a file. The example then reopens the file and reads the data, printing the names.

```
; Create a file to hold the data
file = 'h5_test.h5'
fid = H5F_CREATE(file)

; Create arrays to serve as name/value pairs
names = ['dog', 'pony', 'turtle', 'emu', 'wildebeest']
values = INDGEN(5)+1

; Create the enumeration datatype
dt = H5T_ENUM_CREATE()

; Associate the name/value pairs with the datatype
H5T_ENUM_SET_DATA, dt, names, values

; Create a dataspace, then create and write the dataset
ds = H5S_CREATE_SIMPLE(N_ELEMENTS(values))
d = H5D_CREATE(fid, 'dataset', dt, ds)
H5D_WRITE, d, values

; Close the file
H5F_CLOSE, fid

; Reopen file for reading
fid = H5F_OPEN(file)

; Read in the data
d = H5D_OPEN(fid, 'dataset')
dt = H5D_GET_TYPE(d)
result = H5D_READ(d)

; Close the file
H5F_CLOSE, fid

; Print the value associated with the name "pony"
PRINT, H5T_ENUM_VALUEOF(dt, 'pony')

; Print all the name strings
PRINT, H5T_ENUM_VALUES_TO_NAMES(dt, result)
```

Variable Length Array Datatypes

HDF5 provides support for variable length arrays, but IDL itself does not. As a result, in order to store data in an HDF5 variable length array you must:

1. Create a series of vectors of data in IDL, each with a potentially different length. All vectors must be of the same data type.
2. Store a pointer to each data vector in the PDATA field of an IDL_H5_VLEN structure. The IDL_H5_VLEN structure is defined as follows:

   ```
   { IDL_H5_VLEN, pdata:PTR_NEW() }
   ```

3. Create an array of IDL_H5_VLEN structures that will be stored as an HDF5 variable length array.
4. The IDL_H5_VLEN structure is defined as follows:

   ```
   { IDL_H5_VLEN, pdata:PTR_NEW() }
   ```

5. Create a base HDF5 datatype from one of the data vectors.
6. Create an HDF5 variable length datatype from the base datatype.
7. Create an HDF5 dataspace of the appropriate size.
8. Create an HDF5 dataset.
9. Write the array of IDL_H5_VLEN structures to the HDF5 dataset.

Note

IDL string arrays are a special case: see "Variable Length String Arrays" on page 131 for details.

Creating a variable length array datatype is a two-step process. First, you must create a base datatype using the H5T_IDL_CREATE function; all data in the variable length array must be of this datatype. Second, you create a variable length array datatype using the base datatype as an input to the H5T_VLEN_CREATE function.

Note

No explicit size is provided to the H5T_VLEN_CREATE function; sizes are determined as needed by the data being written.

Example: Writing a Variable Length Array

```
; Create a file to hold the data
file = 'h5_test.h5'
fid = H5F_CREATE(file)
```

```
; Create three vectors containing integer data
a = INDGEN(2)
b = INDGEN(3)
c = 3

; Create an array of three IDL_H5_VLEN structures
sArray = REPLICATE({IDL_H5_VLEN},3)

; Populate the IDL_H5_VLEN structures with pointers to
; the three data vectors
sArray[0].pdata = PTR_NEW(a)
sArray[1].pdata = PTR_NEW(b)
sArray[2].pdata = PTR_NEW(c)

; Create a dataype based on one of the data vectors
dt1 = H5T_IDL_CREATE(a)

; Create a variable length datatype based on the previously-
; created datatype
dt = H5T_VLEN_CREATE(dt1)

; Create a dataspace
ds = H5S_CREATE_SIMPLE(N_ELEMENTS(sArray))

; Create the dataset
d = H5D_CREATE(fid,'dataset', dt, ds)

; Write the array of structures to the dataset
H5D_WRITE, d, sArray
```

Examples: Reading a Variable Length Array

Using the H5D_READ function to read data written as a variable length array creates an array of IDL_H5_VLEN structures. The following examples show how to refer to individual data elements of various HDF5 datatypes

Atomic HDF5 Datatypes

To read and access data stored in variable length arrays of atomic HDF5 datatypes, simply dereference the pointer stored in the PDATA field of the appropriate IDL_H5_VLEN structure. For example, to retrieve the variable `b` from the data written in the above example:

```
data = H5D_READ(d)
b = *data[1].pdata
```

Compound HDF5 Datatypes

If you have a variable length array of compound datatypes, the tag `tag` of the *j*th structure of the *i*th element of the variable length array would be accessed as follows:

```
data = H5D_READ(d)
a = (*data[i].pdata)[j].tag
```

Variable Length Arrays of Variable Length Arrays

If you have a variable length array of variable length arrays of integers, the *k*th integer of the *j*th element of a variable length array stored in the *i*th element of a variable length array would be accessed as follows:

```
data = H5D_READ(d)
a = (*(*data[i].pdata)[j].pdata)[k]
```

Compound Datatypes Containing Variable Length Arrays

If you have a compound datatype containing a variable length array, the *k*th data element of the *j*th variable length array in the *i*th compound datatype would be accessed as follows:

```
data = H5D_READ(d)
a = (*data[i].vl_array[j].pdata)[k]
```

Variable Length String Arrays

Because the data vectors referenced by the pointers stored in the PDATA field of the IDL_H5_VLEN structure must all have the same type and dimension, strings are handled as vectors of individual characters rather than as atomic units. This means that each element in a string array must be assigned to an individual IDL_H5_VLEN structure:

```
str = ['dog', 'dragon', 'duck']
sArray = REPLICATE({IDL_H5_VLEN},3)
sArray[0].pdata = ptr_new(str[0])
sArray[1].pdata = ptr_new(str[1])
sArray[2].pdata = ptr_new(str[2])
```

Use the H5T_STR_TO_VLEN function to assist in converting between an IDL string array and an HDF5 variable length string array. The following achieves the same result as the above five lines:

```
str = ['dog', 'dragon', 'duck']
sArray = H5T_STR_TO_VLEN(str)
```

Similarly, if you have an HDF5 variable length array containing string data, use the H5T_VLEN_TO_STR function to access the string data:

```
data = H5D_READ(d)
str = H5T_VLEN_TO_STR(data)
```

Example: Reading an Image

The following example opens up the `hdf5_test.h5` file and reads in a sample image. It is assumed that the user already knows the dataset name, either from using `h5dump`, or the `H5G_GET_MEMBER_NAME` function.

```
PRO ex_read_hdf5

   ; Open the HDF5 file.
   file = FILEPATH('hdf5_test.h5', $
      SUBDIRECTORY=['examples', 'data'])
   file_id = H5F_OPEN(file)

   ; Open the image dataset within the file.
   ; This is located within the /images group.
   ; We could also have used H5G_OPEN to open up the group first.
   dataset_id1 = H5D_OPEN(file_id, '/images/Eskimo')

   ; Read in the actual image data.
   image = H5D_READ(dataset_id1)

   ; Open up the dataspace associated with the Eskimo image.
   dataspace_id = H5D_GET_SPACE(dataset_id1)

   ; Retrieve the dimensions so we can set the window size.
   dimensions = H5S_GET_SIMPLE_EXTENT_DIMS(dataspace_id)

   ; Now open and read the color palette associated with
   ; this image.
   dataset_id2 = H5D_OPEN(file_id, '/images/Eskimo_palette')
   palette = H5D_READ(dataset_id2)

   ; Close all our identifiers so we don't leak resources.
   H5S_CLOSE, dataspace_id
   H5D_CLOSE, dataset_id1
   H5D_CLOSE, dataset_id2
   H5F_CLOSE, file_id

   ; Display the data.
   DEVICE, DECOMPOSED=0
   WINDOW, XSIZE=dimensions[0], YSIZE=dimensions[1]
   TVLCT, palette[0,*], palette[1,*], palette[2,*]

   ; Use /ORDER since the image is stored top-to-bottom.
   TV, image, /ORDER

END
```

Example: Reading a Subselection

The following example reads only a portion of the previous image, using the dataspace keywords to H5D_READ.

```
PRO ex_read_hdf5_select

   ; Open the HDF5 file.
   file = FILEPATH('hdf5_test.h5', $
      SUBDIRECTORY=['examples', 'data'])
   file_id = H5F_OPEN(file)

   ; Open the image dataset within the file.
   dataset_id1 = H5D_OPEN(file_id, '/images/Eskimo')

   ; Open up the dataspace associated with the Eskimo image.
   dataspace_id = H5D_GET_SPACE(dataset_id1)

   ; Now choose our hyperslab. We will pick out only the central
   ; portion of the image.
   start = [100, 100]
   count = [200, 200]
   ; Be sure to use /RESET to turn off all other
   ; selected elements.
   H5S_SELECT_HYPERSLAB, dataspace_id, start, count, $
      STRIDE=[2, 2], /RESET

   ; Create a simple dataspace to hold the result. If we
   ; didn't supply
   ; the memory dataspace, then the result would be the same size
   ; as the image dataspace, with zeroes everywhere except our
   ; hyperslab selection.
   memory_space_id = H5S_CREATE_SIMPLE(count)

   ; Read in the actual image data.
   image = H5D_READ(dataset_id1, FILE_SPACE=dataspace_id, $
      MEMORY_SPACE=memory_space_id)

   ; Now open and read the color palette associated with
   ; this image.
   dataset_id2 = H5D_OPEN(file_id, '/images/Eskimo_palette')
   palette = H5D_READ(dataset_id2)

   ; Close all our identifiers so we don't leak resources.
   H5S_CLOSE, memory_space_id
   H5S_CLOSE, dataspace_id
   H5D_CLOSE, dataset_id1
   H5D_CLOSE, dataset_id2
   H5F_CLOSE, file_id

   ; Display the data.
   DEVICE, DECOMPOSED=0
   WINDOW, XSIZE=count[0], YSIZE=count[1]
   TVLCT, palette[0,*], palette[1,*], palette[2,*]

   ; We need to use /ORDER since the image is stored
   ; top-to-bottom.
   TV, image, /ORDER

END
```

Example: Creating a Data File

The following example creates a simple HDF5 data file with a single sample data set. The file is created in the current working directory.

```
PRO ex_create_hdf5

   file = filepath('hdf5_out.h5')
   fid = H5F_CREATE(file)

   ;; create data
   data = hanning(100,150)

   ;; get data type and space, needed to create the dataset
   datatype_id = H5T_IDL_CREATE(data)
   dataspace_id = H5S_CREATE_SIMPLE(size(data,/DIMENSIONS))

   ;; create dataset in the output file
   dataset_id = H5D_CREATE(fid,$
      'Sample data',datatype_id,dataspace_id)
   ;; write data to dataset
   H5D_WRITE,dataset_id,data

   ;; close all open identifiers
   H5D_CLOSE,dataset_id
   H5S_CLOSE,dataspace_id
   H5T_CLOSE,datatype_id
   H5F_CLOSE,fid

END
```

Example: Reading Partial Datasets

To read a portion of a compound dataset or attribute, create a datatype that matches only the elements you wish to retrieve, and specify that datatype as the second argument to the H5D_READ function. The following example creates a simple HDF5 data file in the current directory, then opens the file and reads a portion of the data.

```
; Create sample data in an array of structures with two fields
struct = {time:0.0, data:intarr(40)}
r = REPLICATE(struct,20)
r.time = RANDOMU(seed,20)*1000
r.data = INDGEN(40,20)

; Create a file
file = 'h5_test.h5'
fid = H5F_CREATE(file)
; Create a datatype based on a single element of the arrary
dt = H5T_IDL_CREATE(struct)
; Create a 20 element dataspace
ds = H5S_CREATE_SIMPLE(N_ELEMENTS(r))
; Create and write the dataset
d = H5D_CREATE(fid, 'dataset', dt, ds)
H5D_WRITE, d, r
; Close the file
H5F_CLOSE, fid

; Open the file for reading
fid = H5F_OPEN(file)
; Open the dataset
d = H5D_OPEN(fid, 'dataset')
; Define the data we want to read from the dataset
struct = {data:intarr(40)}
; Create datatype denoting the portion to be read
dt = H5T_IDL_CREATE(struct)
; Read only the data that matches our datatype. The
; returned value will be a 20 element structure with only
; one tag, 'DATA'. Each element of which will be a [40]
; element integer array
result = H5D_READ(d, dt)
H5F_CLOSE, fid
```

Alphabetical Listing of HDF5 Routines

H5_CLOSE

H5_CREATE

H5_GET_LIBVERSION

H5_OPEN

H5_PARSE

H5A_CLOSE

H5A_CREATE

H5A_DELETE

H5A_GET_NAME

H5A_GET_NUM_ATTRS

H5A_GET_SPACE

H5A_GET_TYPE

H5A_OPEN_IDX

H5A_OPEN_NAME

H5A_READ

H5A_WRITE

H5D_CLOSE

H5D_CREATE

H5D_EXTEND

H5D_GET_SPACE

H5D_GET_STORAGE_SIZE

H5D_GET_TYPE

H5D_OPEN

H5D_READ

H5D_WRITE

H5F_CLOSE

H5F_CREATE

H5F_IS_HDF5

H5F_OPEN

H5G_CLOSE

H5G_CREATE

H5G_GET_COMMENT

H5G_GET_LINKVAL

H5G_GET_MEMBER_NAME

H5G_GET_NMEMBERS

H5G_GET_NUM_OBJS

H5G_GET_OBJ_NAME_BY_IDX

H5G_GET_OBJINFO

H5G_LINK

H5G_MOVE

H5G_OPEN

H5G_SET_COMMENT

H5G_UNLINK

H5I_GET_FILE_ID

H5I_GET_TYPE

H5R_CREATE

H5R_DEREFERENCE

H5R_GET_OBJECT_TYPE

H5R_GET_REGION

H5S_CLOSE

H5S_COPY

H5S_CREATE_SCALAR

H5S_CREATE_SIMPLE

H5S_GET_SELECT_BOUNDS

H5S_GET_SELECT_ELEM_NPOINTS

H5S_GET_SELECT_ELEM_POINTLIST

H5S_GET_SELECT_HYPER_BLOCKLIST

H5S_GET_SELECT_HYPER_NBLOCKS

H5S_GET_SELECT_NPOINTS

H5S_GET_SIMPLE_EXTENT_DIMS

H5S_GET_SIMPLE_EXTENT_NDIMS

H5S_GET_SIMPLE_EXTENT_NPOINTS

H5S_GET_SIMPLE_EXTENT_TYPE

H5S_IS_SIMPLE

H5S_OFFSET_SIMPLE

H5S_SELECT_ALL

H5S_SELECT_ELEMENTS

H5S_SELECT_HYPERSLAB

H5S_SELECT_NONE

H5S_SELECT_VALID

H5S_SET_EXTENT_NONE

H5S_SET_EXTENT_SIMPLE

H5T_ARRAY_CREATE

H5T_CLOSE

H5T_COMMIT

H5T_COMMITTED

H5T_COMPOUND_CREATE

H5T_COPY

H5T_ENUM_CREATE

H5T_ENUM_GET_DATA

H5T_ENUM_INSERT

H5T_ENUM_NAMEOF

H5T_ENUM_SET_DATA

H5T_ENUM_VALUEOF

H5T_ENUM_VALUES_TO_NAMES

H5T_EQUAL

H5T_GET_ARRAY_DIMS

H5T_GET_ARRAY_NDIMS

H5T_GET_CLASS

H5T_GET_CSET

H5T_GET_EBIAS

H5T_GET_FIELDS

H5T_GET_INPAD

H5T_GET_MEMBER_CLASS

H5T_GET_MEMBER_INDEX

H5T_GET_MEMBER_NAME

H5T_GET_MEMBER_OFFSET

H5T_GET_MEMBER_TYPE

H5T_GET_MEMBER_VALUE

H5T_GET_NMEMBERS

H5T_GET_NORM

H5T_GET_OFFSET

H5T_GET_ORDER

H5T_GET_PAD

H5T_GET_PRECISION

H5T_GET_SIGN

H5T_GET_SIZE

H5T_GET_STRPAD

H5T_GET_SUPER

H5T_GET_TAG

H5T_IDL_CREATE

H5T_IDLTYPE

H5T_INSERT

H5T_MEMTYPE

H5T_OPEN

H5T_REFERENCE_CREATE

H5T_SET_TAG

H5T_STR_TO_VLEN

H5T_VLEN_CREATE

H5T_VLEN_TO_STR

H5_CLOSE

The H5_CLOSE procedure flushes all data to disk, closes file identifiers, and cleans up memory. This routine closes IDL's link to its HDF5 libraries. This procedure is used automatically by IDL when RESET_SESSION is issued, but it may also be called if the user desires to free all HDF5 resources.

Syntax

H5_CLOSE

Arguments

None

Keywords

None

Version History

5.6	Introduced

See Also

H5_OPEN

H5_CREATE

The H5_CREATE function creates and closes a new HDF5 file. This is a simplified routine that encapsulates some of the routines listed in the following sections. Dataspaces are all defined as the full extent of the data, and datatypes are created automatically based on the type of the data.

There are two primary scenarios for the use of H5_CREATE. The first is a new HDF5 file being created from structures created in IDL. The second is an HDF5 file being read using H5_PARSE modifications that are made to the structure with the resulting structure being written to a new file.

Note

Passing the output structure of H5_PARSE to H5_CREATE may not always completely reproduce the original file. Types of things that are not handled by these routines include: references, user-defined datatypes, and the order of items in the file. Additionally, dataset chunking is not supported and thus operations that require chunking are also not supported, for example: dataset extensibility and compression.

Syntax

H5_CREATE, *Filename*, *Structure*

Arguments

Filename

The full path name of the file to create. If the file exists it will be overwritten.

Structure

An IDL structure variable (such as one that could be from H5_PARSE) that conforms to the following:

To create an HDF5 Group the following tags can be used:

Field	Description
_NAME	String: Object name
_TYPE (required)	String: "GROUP" (case insensitive)
_COMMENT	String: Any user defined string
STRUCTURES	Any number of additional structures describing datasets, attributes, groups, or links contained with this group

Table 3-2: H5_CREATE Group Structure Tags

Note

To create a top level group in the file the _NAME field must be defined as the single character /, a null string, or left undefined, otherwise a group underneath the top level group will be created.

To create an HDF5 Dataset the following tags can be used:

Field	Description
_NAME	String: Object name
_TYPE (required)	String: "DATASET" (case insensitive)
_DATA (required)	Any IDL variable (except HDF5 references) accepted by H5D_WRITE
STRUCTURES	Any number of additional structures describing attributes contained with this dataset

Table 3-3: H5_CREATE Dataset Structure Tags

To create an HDF5 Datatype the following tags can be used:

Field	Description
_NAME	String: Object name
_TYPE (required)	String: "DATATYPE" (case insensitive)
_DATA (required)	Any IDL variable (except HDF5 references)
STRUCTURE	Any number of additional structures describing attributes contained within this datatype or describing the individual elements of a compound datatype.

Table 3-4: H5_CREATE Datatype Structure Tags

Note

When creating a DATATYPE structure the _DATA tag is required. However, the structure returned from H5_PARSE can also be used and a proper datatype will be created without the _DATA tag as long as the _DATATYPE, _STORAGESIZE, and _SIGN tags returned are intact. If a compound datatype is being created, and the _DATA tag is not present, the additional structures define the fields of the datatype and the _STORAGESIZE and _SIGN tags are ignored.

To create an HDF5 Attribute the following tags can be used:

Field	Description
_NAME	String: Object name
_TYPE (required)	String: "ATTRIBUTE" (case insensitive)
_DATA (required)	Any IDL variable (except HDF5 references) accepted by H5A_WRITE

Table 3-5: H5_CREATE Attribute Structure Tags

Note

Note: The ATTRIBUTE structure must be contained within at GROUP or DATASET structure, it cannot be a top level structure.

To create an HDF5 Link the following tags can be used:

Field	Description
_NAME	String: Object name
_TYPE (required)	String: "LINK" (case insensitive)
_DATA (required)	(required)String: The name (with path information) of the object to which the link will point
_LINK_TYPE	String: "SOFT" or "HARD" (case insensitive). If not supplied a soft link is created by default

Table 3-6: H5_CREATE Link Structure Tags

Note

The _DATA field must contain the full path information, from the top level group, to the object to which the link will point while _NAME contains the name that will appear in the group in which the link structure exists. For example:

```
{_NAME : "Link1", _TYPE : "LINK", _DATA : "/Group1/MyDataset"}
```

Note

If the _NAME field is not supplied then the name of the structure tag will be used. Additional tags may exist in the structure(s) but will be ignored.

Keywords

None

Examples

As mentioned, there are two primary use cases for H5_CREATE. These are shown in the following example cases.

In the first case, a new HDF5 file is created from structures created in IDL. For example: to create an HDF5 file containing a single data set with a palette attached as an attribute the following could code could be used:

```
grey_scale = byte(bindgen(256)##(bytarr(3)+1b))
```

```
palette = {_TYPE:'Attribute', _DATA:grey_scale}
dataset = {_NAME:'Hanning', _TYPE:'Dataset', $
           _DATA:hanning(100,200), PALETTE:palette}
H5_CREATE, 'myfile.h5', dataset
```

In the second case an HDF5 file is read using H5_PARSE, modifications are made to the structure and the resulting structure is written to a new file. For example, to change the palette in the example file created above so that the colors are reversed:

```
result = H5_PARSE('myfile.h5', /READ_DATA)
newpalette = reverse(result.hanning.palette._data, 2)
result.hanning.palette._data = newpalette
H5_CREATE, 'myNEWfile.h5', result
```

Version History

6.2	Introduced

See Also

H5_PARSE, H5A_WRITE, H5D_WRITE

H5_GET_LIBVERSION

The H5_GET_LIBVERSION function returns the current version of the HDF5 library used by IDL.

Syntax

Result = H5_GET_LIBVERSION( )

Return Value

Returns a string in the form of '*maj*.*min*.*rel*', where *maj* is the major number, *min* is the minor number, and *rel* is the release number. An example would be '1.4.3', representing HDF5 version 1.4.3.

Arguments

None

Keywords

None

Version History

5.6	Introduced

See Also

H5_OPEN

H5_OPEN

The H5_OPEN procedure initializes IDL's HDF5 library. This procedure is issued automatically by IDL when one of IDL's HDF5 routines is used.

Note

This routine is provided for diagnostic purposes only. You do not need to use this routine while working with IDL's HDF5 routines.

Syntax

H5_OPEN

Arguments

None

Keywords

None

Version History

5.6	Introduced

See Also

H5_CLOSE, H5_GET_LIBVERSION

H5_PARSE

The H5_PARSE function recursively descends through an HDF5 file or group and creates an IDL structure containing object information and data.

Note

This function is not part of the standard HDF5 interface, but is provided as a programming convenience.

Two structure fields were added in IDL 6.1. If an H5_PARSE structure from IDL 6.0 is restored the /RELAXED_STRUCTURE_ASSIGNMENT keyword should be used to prevent backward incompatibility.

Syntax

Result = H5_PARSE (*File* [, /READ_DATA])

or

Result = H5_PARSE (*Loc_id*, *Name* [, FILE=*string*] [, PATH=*string*]
[, /READ_DATA] [, /SHOW_HARDLINKS])

Return Value

The *Result* is an IDL structure containing the parsed file or group. The fields within each structure in *Result* depend upon the object type.

Structure Fields Common to All Object Types

Field	Description
_NAME	Object name, or the filename if at the top level
_ICONTYPE	Name of associated icon, used by H5_BROWSER
_TYPE	Object type, such as GROUP, DATASET, DATATYPE, ATTRIBUTE, or LINK

Table 3-7: Structure Fields Common to All Object Types

Additional Fields for Groups, Datasets, and Named Datatypes

Field	Description
_FILE	The filename to which the object belongs
_PATH	Full path to the group, dataset, or datatype within the file

Table 3-8: Additional Fields for Groups, Datasets, and Named Datatypes

Additional Fields for Groups

Field	Description
_COMMENT	Comment string

Table 3-9: Additional Fields for Groups

Additional Fields for Datasets, Attributes, and Named Datatypes

Field	Description
_DATATYPE	Datatype class, such as H5T_INTEGER
_STORAGESIZE	Size of each value in bytes

Table 3-10: Additional Fields for Datasets, Attributes, and Named Datatypes

Field	Description
_PRECISION	Precision of each value in bits
_SIGN	For integers, either ‘signed’ or ‘unsigned’; otherwise a null string

Table 3-10: Additional Fields for Datasets, Attributes, and Named Datatypes (Continued)

Additional Fields for Datasets and Attributes

Field	Description
_DATA	Data values stored in the object
_NDIMENSIONS	Number of dimensions in the dataspace
_DIMENSIONS	List of dataspace dimensions
_NELEMENTS	Total number of elements in the dataspace
_HARDLINK	Full path to the object being linked to. If this is not a null string then the current object is actually a hard link to the object denoted by this string.

Table 3-11: Additional Fields for Datasets and Attributes

Additional Fields for Links

Field	Description
_LINKTYPE	If the SHOW_HARDLINKS keyword is set then this field will be added to links and will contain the value ‘HARD’. Soft links will not have this field added.

Table 3-12: Additional Fields Links

Groups, datasets, datatypes, and attributes will be stored as substructures within *Result*. The tag names for these substructures are constructed from the actual object name by converting all non-alphanumeric characters to underscores, and converting all characters to uppercase. If a tag name already exists (for example a datatype and an attribute have the same name) then an appropriate suffix is appended on to the end of the tag name, such as "_ATTR" for attribute, and so on.

If a tag name already exists within the same dataset then the suffix that is appended on to the end of the tag name will consist of _X where X starts with 1 and increments as needed.

Arguments

File

A string giving the name of the file to parse.

Loc_id

An integer giving the file or group identifier to access.

Name

A string giving the name of the group, dataset, or datatype within *Loc_id* to parse.

Keywords

FILE

Set this optional keyword to a string giving the filename associated with the Loc_id. This keyword is used for filling in the _FILE field within the returned structure, and is not required. The FILE keyword is ignored if the *File* argument is provided.

PATH

Set this optional keyword to a string giving the full path associated with the *Loc_id*. This keyword is used for filling in the _PATH field within the returned structure, and is not required. The PATH keyword is ignored if the *File* argument is provided.

READ_DATA

If this keyword is set, then all data from datasets is read in and stored in the returned structure. If READ_DATA is not provided then the _DATA field for datasets will be set to the string ‘<unread>’.

Note

For attribute objects all data is automatically read and stored in the structure.

SHOW_HARDLINKS

If this keyword is set, then hardlinks will appear as a LINK structure. The default is to treat hardlinks as copies of the object pointed to.

Note

Because there is no distinguishable difference between a hard link and the object to which the link points, the first object encountered in the file is taken to be the object and any subsequent apparent copies of the object are taken to be links. This may be different than the actual order in the file.

Example

The following example shows how to parse a file, and then prints out the parsed structure.

```
File = FILEPATH('hdf5_test.h5', SUBDIR=['examples','data'])
Result = H5_PARSE(File)
help, Result, /STRUCTURE
```

When the above commands are entered, IDL prints:

```
** Structure <5f24468>, 13 tags, length=6872, data length=6664,
refs=1:
   _NAME STRING    'D:\RSI\idl63\examples\data\hdf5_test.h5'
   _ICONTYPE       STRING    'hdf'
   _TYPE       STRING    'GROUP'
   _FILE       STRING    'D:\RSI\idl63\examples\data\hdf5_test.h5'
   _PATH       STRING    '/'
   _COMMENT        STRING    ''
   _2D_INT_ARRAY   STRUCT    -> <Anonymous> Array[1]
   A_NOTE          STRUCT    -> <Anonymous> Array[1]
   SL_TO_3D_INT_ARRAY
   STRUCT    -> <Anonymous> Array[1]
   ARRAYS          STRUCT    -> <Anonymous> Array[1]
   DATATYPES       STRUCT    -> <Anonymous> Array[1]
   IMAGES          STRUCT    -> <Anonymous> Array[1]
   LINKS           STRUCT    -> <Anonymous> Array[1]
```

Now print out the structure of a dataset within the “Images” group:

```
help, Result.images.eskimo, /STRUCTURE
```

IDL prints:

```
** Structure <16f1ca0>, 20 tags, length=840, data length=802,
refs=2:
   _NAME            STRING    'Eskimo'
   _ICONTYPE        STRING    'binary'
   _TYPE            STRING    'DATASET'
   _FILE            STRING
'D:\RSI\debug\examples\data\hdf5_test.h5'
   _PATH            STRING    '/images'
   _DATA            STRING    '<unread>'
   _NDIMENSIONS     LONG                     2
   _DIMENSIONS      ULONG64   Array[2]
   _NELEMENTS       ULONG64                       389400
   _DATATYPE        STRING    'H5T_INTEGER'
   _STORAGESIZE     ULONG                    1
   _PRECISION       LONG                     8
   _SIGN            STRING    'unsigned'
   CLASS            STRUCT    -> <Anonymous> Array[1]
   IMAGE_VERSION    STRUCT    -> <Anonymous> Array[1]
   IMAGE_SUBCLASS   STRUCT    -> <Anonymous> Array[1]
   IMAGE_COLORMODEL
                    STRUCT    -> <Anonymous> Array[1]
   IMAGE_MINMAXRANGE
                    STRUCT    -> <Anonymous> Array[1]
   IMAGE_TRANSPARENCY
                    STRUCT    -> <Anonymous> Array[1]
   PALETTE          STRUCT    -> <Anonymous> Array[1]
```

Version History

5.6	Introduced
6.2	Added _HARDLINK and _LINKTYPE structure fields.

See Also

H5_BROWSER

H5A_CLOSE

The H5A_CLOSE procedure closes the specified attribute and releases resources used by it. After this routine is used, the attribute's identifier is no longer available until the H5A_OPEN routines are used again to specify that attribute. Further use of the attribute identifier is illegal.

Syntax

H5A_CLOSE, *Attribute_id*

Arguments

Attribute_id

An integer representing the attribute's identifier to be closed.

Keywords

None

Version History

5.6	Introduced

See Also

H5A_OPEN_NAME, H5A_OPEN_IDX

H5A_CREATE

The H5A_CREATE function creates a dataset as an attribute of another group or dataset.

Note

Attributes are intended to be small objects with a maximum size of 16 kilobytes, data sizes greater than this limit will cause the attribute creation to fail. A large dataset intended as meta data for another dataset can be stored as an additional dataset. An attribute can then be attached to the original dataset as an object reference pointer to the desired supplemental dataset.

Syntax

Result = H5A_CREATE(*Loc_id*, *Name*, *Datatype_id*, *Dataspace_id*)

Return Value

The *Result* gives the attribute identifier number. This identifier should be released with the H5A_CLOSE procedure.

Arguments

Loc_id

An integer giving the identifier of the group, dataset, or named datatype to which the attribute will be attached

Name

A string giving the name of the attribute to create.

Datatype_id

An integer giving the datatype identifier of the new attribute.

Dataspace_id

An integer giving the dataspace identifier of the new attribute.

Keywords

None

Example

See the example under H5_CREATE.

Version History

6.2	Introduced

See Also

H5A_CLOSE, H5S_CREATE_SIMPLE, H5T_IDL_CREATE

H5A_DELETE

The H5A_DELETE procedure removes the attribute specified by its name from a dataset, group, or named datatype.

Note

This function requires that all attributes be closed on the specified object and will close any attributes currently open.

Syntax

H5A_DELETE, *Loc_id*, *Name*

Arguments

Loc_id

An integer giving the identifier of the group, dataset, or named datatype from which the attribute will be deleted.

Name

A string giving the name of the attribute to delete.

Keywords

None

Example

See the example under H5F_CREATE.

Version History

6.2	Introduced

See Also

H5A_CREATE

H5A_GET_NAME

The H5A_GET_NAME function retrieves an attribute name given the attribute identifier number.

Syntax

Result = H5A_GET_NAME(*Attribute_id*)

Return Value

Returns a string containing the attribute name.

Arguments

Attribute_id

An integer representing the attribute’s identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5A_GET_SPACE, H5A_GET_TYPE

H5A_GET_NUM_ATTRS

The H5A_GET_NUM_ATTRS function returns the number of attributes attached to a group, dataset, or a named datatype.

Syntax

Result = H5A_GET_NUM_ATTRS(*Loc_id*)

Return Value

Returns the number of attributes.

Arguments

Loc_id

An integer representing the identifier of the group, dataset, or named datatype to query.

Keywords

None

Version History

5.6	Introduced

See Also

H5A_OPEN_IDX

H5A_GET_SPACE

The H5A_GET_SPACE function returns the identifier number of a copy of the dataspace for an attribute.

Syntax

Result = H5A_GET_SPACE(*Attribute_id*)

Return Value

Returns the dataspace's identifier. This identifier can be released with the H5S_CLOSE.

Arguments

Attribute_id

An integer representing the attribute's identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5A_GET_NAME, H5A_GET_TYPE, H5S_CLOSE

H5A_GET_TYPE

The H5A_GET_TYPE function returns the identifier number of a copy of the datatype for an attribute.

Syntax

Result = H5A_GET_TYPE(*Attribute_id*)

Return Value

Returns the datatype identifier. This identifier should be released with the H5T_CLOSE.

Arguments

Attribute_id

An integer representing the attribute identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5A_GET_SPACE, H5A_GET_NAME, H5T_CLOSE

H5A_OPEN_IDX

The H5A_OPEN_IDX function opens an existing attribute by the index of that attribute within an HDF5 file.

Syntax

Result = H5A_OPEN_IDX(*Loc_id*, *Index*)

Return Value

Returns the attribute's identifier number.

Arguments

Loc_id

An integer representing the identifier of the group, dataset, or named datatype containing the attribute within.

Index

An integer representing the zero-based index of the attribute to be accessed.

Keywords

None

Version History

5.6	Introduced

See Also

H5A_OPEN_NAME, H5A_GET_NUM_ATTRS, H5A_GET_NAME, H5A_CLOSE

H5A_OPEN_NAME

The H5A_OPEN_NAME function opens an existing attribute by the name of that attribute within an HDF5 file.

Syntax

Result = H5A_OPEN_NAME(*Loc_id*, *Name*)

Return Value

Returns the attribute's identifier number.

Arguments

Loc_id

An integer representing the identifier of the group, dataset, or named datatype containing the attribute within.

Name

A string representing the name of the attribute to be accessed.

Keywords

None

Version History

5.6	Introduced

See Also

H5A_OPEN_IDX, H5A_CLOSE

H5A_READ

The H5A_READ function reads the data within an attribute, converting from the HDF5 file datatype into the HDF5 memory datatype, and finally into the corresponding IDL datatype.

Syntax

Result = H5A_READ(*Attribute_id* [, *Datatype_id*])

Return Value

Returns an IDL variable containing all of the attribute's data. For details on different return types and storage mechanisms, see the H5D_READ function.

Arguments

Attribute_id

A long integer containing the identifier of the attribute to read.

Datatype_id

A long integer containing the identifier of the memory datatype to read. This argument is used only when reading part of a compound attribute. If *Datatype_id* is not supplied, the entire attribute is read.

Keywords

None

Version History

5.6	Introduced
6.3	Added the *Datatype_id* argument.

See Also

H5A_OPEN_NAME, H5A_OPEN_IDX, H5A_CLOSE, H5D_READ

H5A_WRITE

The H5A_WRITE procedure writes data to an attribute.

Syntax

H5A_WRITE, *Attribute_id*, *Data*

Arguments

Attribute_id

An integer giving the identifier of the attribute to which to write the data.

Data

The data to be written. The following table shows how IDL data types are converted to HDF5 datatypes. Pointers, complex numbers, and object references cannot be written to HDF5 attributes. Data passed in via IDL will automatically be converted into the output data type if possible.

IDL Data Type	HDF5 Data Type
Byte	H5T_NATIVE_UINT8
Integer	H5T_NATIVE_INT16
Unsigned integer	H5T_NATIVE_UINT16
Long integer	H5T_NATIVE_INT32
Unsigned long integer	H5T_NATIVE_UINT32
64-bit Integer	H5T_NATIVE_INT64
Unsigned 64-bit integer	H5T_NATIVE_UINT64
Floating point	H5T_NATIVE_FLOAT
Double-precision floating	H5T_NATIVE_DOUBLE
String	H5T_C_S1

Table 3-13: IDL to HDF5 Corresponding Data Types

IDL Data Type	HDF5 Data Type
Reference Structure	H5T_REFERENCE
Structure	(Member datatypes)

Table 3-13: IDL to HDF5 Corresponding Data Types (Continued)

Note

The reference structure is returned from H5R_CREATE.

Keywords

None

Example

See the example under H5F_CREATE.

Version History

6.2	Introduced

See Also

H5A_CREATE, H5S_CREATE_SIMPLE, H5T_IDL_CREATE, H5T_REFERENCE_CREATE

H5D_CLOSE

The H5D_CLOSE procedure closes the specified dataset and releases its used resources. After this routine is used, the dataset's identifier is no longer available until the H5D_GET_SPACE is used again to specify that dataset.

Syntax

H5D_CLOSE, *Dataset_id*

Arguments

Dataset_id

An integer representing the dataset's identifier to be closed.

Keywords

None

Version History

5.6	Introduced

See Also

H5D_OPEN

H5D_CREATE

The H5D_CREATE function creates a dataset at the specified location.

Syntax

Result = H5D_CREATE(*Loc_id*, *Name*, *Datatype_id*, *Dataspace_id*
[, CHUNK_DIMENSIONS=*vector* [, GZIP=*value* [, /SHUFFLE]]])

Return Value

The *Result* gives the dataset identifier. This identifier should be released with the H5D_CLOSE procedure.

Arguments

Loc_id

An integer giving the identifier of the file or group within which to create the dataset.

Name

A string giving the name of the dataset to create.

Datatype_id

An integer giving the datatype identifier to use when creating the dataset.

Dataspace_id

An integer giving the dataspace identifier to use when creating the dataset.

Keywords

CHUNK_DIMENSIONS

A vector containing the chunk dimensions for the dataset. CHUNK_DIMENSIONS must have the same number of elements as the number of dimensions in the dataspace specified in Dataspace_id. This keyword must be set if the dataspace specified in Dataspace_id has unlimited or extendable dimensions.

Note

Choosing appropriate values for CHUNK_DIMENSIONS is not always straight forward and is dependant on the size of the dataspace, the size of the data, how the data will be read, the current operating system, and many other factors. Improper chunk sizes can drastically inflate the size of the resulting file or greatly slow the reading of the data. For a dimension that is immutable a good suggestion is to choose a value that is evenly divisible into the dimension size. Values of less than 100 for dataspaces with dimensions greater than 1000 can result in bloated file sizes.

GZIP

Specifies the level of gzip compression applied to the dataset, which should be a value from zero to nine, inclusive. Lower compression levels are faster but result in less compression. If CHUNK_DIMENSIONS is not specified this keyword is ignored.

SHUFFLE

If set the shuffle filter will be applied to the dataset. If GZIP is not specified this keyword is ignored.

The shuffle filter de-interlaces a block of data by reordering the bytes. All bytes from one consistent byte position of each data element are placed together in one block; all bytes from a second consistent byte position of each data element are placed together a second block; and so on. For example, given three data elements of a 4-byte datatype stored as 012301230123, shuffling will re-order data as 000111222333. This can be a valuable step in an effective compression algorithm because the bytes in each byte position are often closely related to each other and putting them together can increase the compression ratio. When the shuffle filter is applied to a dataset, the compression ratio achieved is often superior to that achieved without the shuffle filter.

Example

See the example under H5F_CREATE.

Version History

6.2	Introduced

See Also

H5D_CLOSE, H5S_CREATE_SIMPLE, H5T_IDL_CREATE

H5D_EXTEND

The Dataspace of a dataset defines the number of dimensions and the size of each dimension. H5D_EXTEND is used to change the current dimensions of the Dataset, within the limits of the Dataspace. Each dimension can be extended up to its maximum, or unlimited. The maximum dimension size is set when the Dataset is created and cannot be changed. The size of the dataset cannot be reduced after it is created. The actual dimension size can be incremented with calls to H5D_EXTEND, up to the maximum.

Syntax

H5D_EXTEND,*Dataset_id*, *Size*

Arguments

Dataset_id

An integer giving the dataset identifier to extend.

Size

Array containing the new magnitude of each dimension. The number of elements in Size must match the number of dimensions of the dataset.

Note

The Size argument should be specified in IDL column-major order. Internally, the dimensions will be reversed to match HDF5/C row-major order.

Keywords

None

Example

See the example under H5F_CREATE.

Version History

6.2	Introduced

See Also

H5D_CREATE

H5D_GET_SPACE

The H5D_GET_SPACE function returns an identifier number for a copy of the dataspace for a dataset.

Syntax

Result = H5D_GET_SPACE(*Dataset_id*)

Return Value

Returns the dataspace's identifier. This identifier can be released with the H5S_CLOSE.

Arguments

Dataset_id

An integer representing the dataset's identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5S_CLOSE, H5D_GET_STORAGE_SIZE, H5D_GET_TYPE

H5D_GET_STORAGE_SIZE

The H5D_GET_STORAGE_SIZE function returns the amount of storage in bytes required for a dataset. For chunked datasets, this value is the number of allocated chunks times the chunk size.

Note

This function does not typically need to be called, as IDL will automatically allocate the necessary memory when reading data.

Syntax

Result = H5D_GET_STORAGE_SIZE(*Dataset_id*)

Return Value

Returns the amount of storage in bytes.

Arguments

Dataset_id

An integer representing the dataset's identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5S_CLOSE, H5D_GET_SPACE, H5D_GET_TYPE

H5D_GET_TYPE

The H5D_GET_TYPE function returns an identifier number for a copy of the datatype for a dataset.

Syntax

Result = H5D_GET_TYPE(*Dataset_id*)

Return Value

Returns the datatype’s identifier. This identifier can be released with the H5T_CLOSE.

Arguments

Dataset_id

An integer representing the dataset’s identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_CLOSE, H5D_GET_SPACE, H5D_GET_STORAGE_SIZE

H5D_OPEN

The H5D_OPEN function opens an existing dataset within an HDF5 file.

Syntax

Result = H5D_OPEN(*Loc_id*, *Name*)

Return Value

Returns the dataset's identifier. This identifier can be released with the H5D_CLOSE.

Arguments

Loc_id

An integer representing the identifier of the file or group containing the dataset.

Name

A string representing the name of the dataset to be accessed.

Keywords

None

Version History

5.6	Introduced

See Also

H5D_CLOSE

H5D_READ

The H5D_READ function reads the data within a dataset, converting from the HDF5 file datatype into the HDF5 memory datatype, and finally into the corresponding IDL datatype.

Syntax

Result = H5D_READ(*Dataset_id* [, *Datatype_id*] [, FILE_SPACE=*id*]
[, MEMORY_SPACE=*id*])

Return Value

Returns an IDL variable containing the specified data. The different return types and storage mechanisms are described below.

Note

The dimensions for the *Result* variable are constructed using the following precedence rules:

If MEMORY_SPACE is specified, then the dimensions of the MEMORY_SPACE are used.

If only FILE_SPACE is specified, then the dimensions of the FILE_SPACE are used.

If neither MEMORY_SPACE nor FILE_SPACE are specified, then the dimensions are taken from the *Dataset_id*.

Arguments

Dataset_id

A long integer containing the identifier of the dataset to read.

Datatype_id

A long integer containing the identifier of the memory datatype to read. This argument is used only when reading part of a compound dataset. If *Datatype_id* is not supplied, the entire dataset is read.

Keywords

FILE_SPACE

Set this keyword to the file dataspace identifier that should be used when reading the dataset. The FILE_SPACE keyword may be used to define hyperslabs or elements for subselection within the dataset. The default is zero (in HDF5 this is equivalent to H5S_ALL), which indicates that the entire dataspace should be read.

MEMORY_SPACE

Set this keyword to the memory dataspace identifier that should be used when copying the data from the file into memory. The MEMORY_SPACE keyword may be used to define hyperslabs or elements in which to place the data. The default is zero (in HDF5 this is equivalent to H5S_ALL), which indicates that the memory dataspace is identical to the file dataspace.

Return Type

When reading in HDF5 datasets, the datatype is first set to the native HDF5 types. These native types are then converted to IDL types as shown in the following table:

HDF5 Class	HDF5 Datatype	IDL Type
H5T_INTEGER H5T_BITFIELD H5T_ENUM	H5T_NATIVE_UINT8	Byte
	H5T_NATIVE_INT16	Integer
	H5T_NATIVE_UINT16	Unsigned integer
	H5T_NATIVE_INT32	Long integer
	H5T_NATIVE_UINT32	Unsigned long integer
	H5T_NATIVE_INT64	64-bit integer
	H5T_NATIVE_UINT64	Unsigned 64-bit integer
H5T_REFERENCE	H5T_STD_REF_OBJ	Unsigned 64-bit integer
	H5T_REF_DSETREG	Structure

Table 3-14: HDF and IDL Datatypes

HDF5 Class	HDF5 Datatype	IDL Type
H5T_FLOAT	H5T_NATIVE_FLOAT	Floating point
	H5T_NATIVE_DOUBLE	Double-precision floating
H5T_STRING	H5T_C_S1	String
H5T_TIME	H5T_C_S1	String
H5T_COMPOUND	(Member datatypes)	Structure
H5T_ARRAY	(Super datatype)	(Super type)

Table 3-14: HDF and IDL Datatypes (Continued)

Note

Multidimensional datasets are returned in IDL column-major order, with the fastest-varying dimensions listed first. HDF5 uses C row-major order, with the fastest-varying dimensions listed last. In both cases, the memory layout for data elements is identical (i.e., no transpose is needed), and only the order of the dimensions is reversed.

Note

For the H5T_ARRAY datatype, the array dimensions are concatenated with the dataset dimensions, with the array dimensions varying more rapidly.

Note

Structure tag names are constructed from H5T_COMPOUND member names by switching to uppercase and converting all non-alphanumeric characters to underscores.

Version History

5.6	Introduced
6.2	Added H5T_STD_REF_DSETREG Datatype (structure IDL type) to the H5T_REFERENCE class
6.3	Added the *Datatype_id* argument

See Also

H5D_CLOSE, H5D_OPEN, H5A_READ, H5S_CREATE_SIMPLE, H5S_SELECT_ELEMENTS, H5S_SELECT_HYPERSLAB

H5D_WRITE

The H5D_WRITE procedure writes data to a dataset.

Syntax

H5D_WRITE, *Dataset_id*, *Data* [, MEMORY_SPACE_ID=*value*]
[, FILE_SPACE_ID=*value*]

Arguments

Dataset_id

An integer giving the dataspace identifier to which to write the data.

Data

The data containing the selection to be written. The table shows how IDL data types are converted to HDF5 datatypes. Pointers and object references cannot be written to HDF5 datasets. Data passed in via IDL will automatically be converted into the output data type if possible.

Keywords

MEMORY_SPACE_ID

An integer giving the identifier of the dataspace of the dataset. The default is to use the entire dataset.

FILE_SPACE_ID

An integer giving the identifier of dataset's dataspace in the file. The default is to use the entire dataset.

Example

See the example under H5F_CREATE.

Version History

6.2	Introduced

See Also

H5D_CREATE, H5S_CREATE_SIMPLE

H5F_CLOSE

The H5F_CLOSE procedure closes the specified file and releases resources used by it. After this routine is used, the file's identifier is no longer available.

Syntax

H5F_CLOSE, *File_id*

Arguments

File_id

An integer representing the file's identifier to be closed.

Keywords

None

Version History

5.6	Introduced

See Also

H5F_OPEN

H5F_CREATE

The H5F_CREATE function is the primary function for creating HDF5 files.

Note

As an alternative, see H5_CREATE.

Syntax

Result = H5F_CREATE(*Filename*)

Return Value

Result is a file identifier for the newly-created file; this file identifier should be closed by calling H5F_CLOSE when it is no longer needed.

Arguments

Filename

A string giving the name of the file to create.

Keywords

None

Example

```
; create HDF5 file
file = 'hdf5_out.h5'
fid = H5F_CREATE(file)

; create some data
data = hanning(100,200)

; create a datatype
datatype_id = H5T_IDL_CREATE(data)

; create a dataspace, allow the dataspace to be extendable
dataspace_id = $
   H5S_CREATE_SIMPLE([100,100],max_dimensions=[200,200])
```

```
; create the dataset
dataset_id = H5D_CREATE(fid,'Hanning',datatype_id,dataspace_id, $
   chunk_dimensions=[20,20])

; extend the size of the dataset to fit the data
H5D_EXTEND,dataset_id,size(data,/dimensions)

; write the data to the dataset
H5D_WRITE,dataset_id,data

; close some identifiers
H5S_CLOSE,dataspace_id
H5T_CLOSE,datatype_id

; create a reference attribute attached to the dataset
dataspace_id = H5S_CREATE_SIMPLE(size(data,/dimensions))

; select a 30x30 element region of interest in the dataset
H5S_SELECT_HYPERSLAB,dataspace_id,[40,40],[1,1], $
   block=[30,30],/reset

; create a dataspace region reference
ref = H5R_CREATE(fid,'Hanning',dataspace=dataspace_id)

; create a datatype for the reference
datatype_id = H5T_REFERENCE_CREATE(/region)

; create a one element dataspace for the single reference
dataspace_id = H5S_CREATE_SIMPLE(1)

; make the reference an attribute of the dataset
attr_id = H5A_CREATE(dataset_id,'Ref',datatype_id,dataspace_id)
H5A_WRITE,attr_id,ref
H5A_CLOSE,attr_id

; create a dummy attribute and delete it
attr_id2 = $
H5A_CREATE(dataset_id,'Dummy',datatype_id,dataspace_id)

; attribute must be closed before it can be deleted
H5A_CLOSE,attr_id2
H5A_DELETE,dataset_id,'Dummy'

; create a group to hold sample datatypes and links
group_id = H5G_CREATE(fid,'Datatypes and links')

; add a comment to the group
H5G_SET_COMMENT,fid,'Datatypes and links', $
   'This is a sample comment'
```

```
; add a datatype to the group
datatype_id2 = H5T_IDL_CREATE(1)

; add the datatype to the group and give it a name
H5T_COMMIT,group_id,'Integer',datatype_id2

; create an array datatype and add it to the group with a name
datatype_id3 = H5T_ARRAY_CREATE(datatype_id2,[3,4])
H5T_COMMIT,group_id,'Integer 2',datatype_id3

; rename previous datatype
H5G_MOVE,group_id,'Integer 2','Integer Array'

; close temporary datatypes
H5T_CLOSE,datatype_id3
H5T_CLOSE,datatype_id2

; create a compound datatype and add it to the group
struct = {float:1.0, double:1.0d}
datatype_id4 = $
   H5T_IDL_CREATE(struct,member_names=['Float','Double'])

; create an integer datatype and insert it in the
; compound datatype
datatype_id5 = H5T_IDL_CREATE(1)
H5T_INSERT,datatype_id4,'Integer',datatype_id5

; add the datatype to the group and give it a name
H5T_COMMIT,group_id,'Compound',datatype_id4

; close datatype identifiers
H5T_CLOSE,datatype_id5
H5T_CLOSE,datatype_id4

; add a hard link from the group to the Hanning dataset
H5G_LINK,fid,'Hanning','Link to Hanning',new_loc_id=group_id

; add a dummy link
H5G_LINK,group_id,'Integer','Link to Integer'

; remove dummy link
H5G_UNLINK,group_id,'Link to Integer'

; close remaining open identifiers
H5G_CLOSE,group_id
H5D_CLOSE,dataset_id
H5T_CLOSE,datatype_id
H5S_CLOSE,dataspace_id
H5F_CLOSE,fid
```

Version History

6.2	Introduced

See Also

H5F_CLOSE

H5F_IS_HDF5

The H5F_IS_HDF5 function determines if a file is in the HDF5 format.

Syntax

Result = H5F_IS_HDF5(*Filename*)

Return Value

Returns 1 if the file is in the HDF5 format, 0 if otherwise.

Arguments

Filename

A string representing the name of the files to be checked.

Keywords

None

Version History

5.6	Introduced

See Also

H5F_OPEN

H5F_OPEN

The H5F_OPEN function opens an existing HDF5 file.

Syntax

Result = H5F_OPEN(*Filename*)([, /WRITE])

Return Value

Returns the file identifier number. This identifier can be released with the H5F_CLOSE.

Arguments

Filename

A string representing the name of the file to be accessed.

Keywords

WRITE

If set the file is open for both reading and writing. The default is to open the file in read_only mode.

Version History

5.6	Introduced
6.2	Added WRITE keyword.

See Also

H5F_CLOSE, H5F_IS_HDF5

H5G_CLOSE

The H5G_CLOSE procedure closes the specified group and releases resources used by it. After this routine is used, the group's identifier is no longer available until the H5F_OPEN routine is used again to specify that group.

Syntax

H5G_CLOSE, *Group_id*

Arguments

Group_id

An integer representing the group's identifier to be closed.

Keywords

None

Version History

5.6	Introduced

See Also

H5G_OPEN

H5G_CREATE

The H5G_CREATE function creates a new empty group and gives it a name.

Syntax

Result = H5G_CREATE(*Loc_id*, *Name*)

Return Value

Result is the group identifier for the open group; this group identifier should be closed by calling H5G_CLOSE when it is no longer needed.

Arguments

Loc_id

An integer giving the identifier of the file or group.

Name

A string giving the name of the new group.

Keywords

None

Example

See the example under H5F_CREATE.

Version History

6.2	Introduced

See Also

H5G_CLOSE

H5G_GET_COMMENT

The H5G_GET_COMMENT function retrieves a comment string from a specified object.

Syntax

Result = H5G_GET_COMMENT(*Loc_id*, *Name*)

Return Value

Returns a string containing the comment, or a null string if no comment exists.

Arguments

Loc_id

An integer representing the identifier of the file or group.

Name

A string representing the name of the object for which to retrieve the comment.

Keywords

None

Version History

5.6	Introduced

See Also

H5G_GET_OBJINFO

H5G_GET_LINKVAL

The H5G_GET_LINKVAL function returns the name of the object pointed to by a symbolic link.

Syntax

Result = H5G_GET_LINKVAL(*Loc_id*, *Name*)

Return Value

Returns a string containing the name of the object pointed to by a symbolic link.

Arguments

Loc_id

An integer representing the identifier of the file or group.

Name

A string representing the name of the symbolic link for which to retrieve the link value.

Keywords

None

Version History

5.6	Introduced

See Also

H5G_GET_OBJINFO

H5G_GET_MEMBER_NAME

The H5G_GET_MEMBER_NAME function retrieves the name of an object within a group, by its zero-based index.

Note

This function is not part of the standard HDF5 interface, but is provided as a programming convenience. The H5Giterate() C function is used to retrieve the name.

Syntax

Result = H5G_GET_MEMBER_NAME(*Loc_id*, *Name*, *Index*)

Return Value

Returns a string containing the object's name.

Arguments

Loc_id

An integer representing the identifier of the file or group.

Name

A string representing the name of the group in which to retrieve the member name.

Index

An integer representing the zero-based index of the object for which to retrieve the name.

Keywords

None

Version History

5.6	Introduced

See Also

H5G_GET_NMEMBERS

H5G_GET_NMEMBERS

The H5G_GET_NMEMBERS function returns the number of objects within a group.

Note

This function is not part of the standard HDF5 interface, but is provided as a programming convenience. The H5Giterate() C function is used to retrieve the number of members.

Syntax

Result = H5G_GET_NMEMBERS(*Loc_id*, *Name*)

Return Value

Returns the number of objects.

Arguments

Loc_id

An integer representing the identifier of the file or group.

Name

A string representing the name of the group for which to retrieve the number of members.

Keywords

None

Version History

5.6	Introduced

See Also

H5G_GET_MEMBER_NAME

H5G_GET_NUM_OBJS

The H5G_GET_NUM_OBJS function returns number of objects in the group specified by its identifier.

Syntax

Result = H5G_GET_NUM_OBJS(*Loc_id*)

Return Value

Result is the number of objects contained in the group.

Arguments

Loc_id

An integer giving the file or group identifier.

Keywords

None

Version History

6.2	Introduced

See Also

H5G_CREATE, H5G_GET_OBJ_NAME_BY_IDX

H5G_GET_OBJ_NAME_BY_IDX

The H5G_GET_OBJ_NAME_BY_IDX function returns a name of an object specified by an index.

Syntax

Result = H5G_GET_OBJ_NAME_BY_IDX(*Loc_id*, *Index*)

Return Value

Result is a string containing the name of the object.

Arguments

Loc_id

An integer giving file or group identifier.

Index

An integer index identifying the object.

Keywords

None

Version History

6.2	Introduced

See Also

H5G_CREATE, H5G_GET_NUM_OBJS

H5G_GET_OBJINFO

The H5G_GET_OBJINFO function retrieves information from a specified object.

Syntax

Result = H5G_GET_OBJINFO(*Loc_id*, *Name* [, /FOLLOW_LINK])

Return Value

Returns a structure of the name H5F_STAT containing the following fields:

FILENO

This field contains two integers which, along with the OBJNO field, uniquely identify the object among all open HDF5 files.

OBJNO

This field contains two integers which, along with the FILENO field, uniquely identify the object among all open HDF5 files. If all four values in FILENO and OBJNO are the same between two objects, then these two objects are the same.

NLINK

The number of hard links to the object. If this field is zero, then the object is a symbolic link.

TYPE

A string representing the object type. Possible values are:

- ‘LINK’
- ‘GROUP’
- ‘DATASET’
- ‘TYPE’
- ‘UNKNOWN’

MTIME

The modification time for the object, in seconds since 1 January 1970.

Tip

You can convert the MTIME field from seconds to a date/time string using SYSTIME(*0*, *mtime*). See SYSTIME for more information.

LINKLEN

If the object is a symbolic link (and the FOLLOW_LINK keyword is not set), then this field will contain the length in characters of the link value. The link value itself may be retrieved using H5D_GET_LINKVAL.

Arguments

Loc_id

An integer representing the identifier of the file or group.

Name

A string representing the name of the object for which to retrieve the information structure.

Keywords

FOLLOW_LINK

If *Name* is a symbolic link, then set this keyword to follow the symbolic link and retrieve information about the linked object. The default is to return information about the symbolic link itself.

Version History

5.6	Introduced

See Also

H5G_GET_LINKVAL

H5G_LINK

The H5G_LINK procedure creates a link of the specified type. A link can only point to one of the three classes of named objects: group, dataset, and named datatype.

Syntax

H5G_LINK, *Loc_id*, *Current_Name*, *New_Name* [, /SOFTLINK]
[, NEW_LOC_ID=*value*]

Arguments

Loc_id

An integer giving the file or group identifier.

Current_Name

String name of the existing object if link is a hard link. Can be anything for the soft link.

When creating a soft link *Current_Name* can be absolute or relative and may include path information.

For example, to create a link to an object that exists in the current group use the name of the object:

```
Object1
```

To create a link to an object that exists in a sub group of the current group use a relative path name:

```
Subgroup/Object1 or ./Subgroup/Object1
```

To create a link to an object that exists outside of the current group use an absolute path (a path beginning with the root group of the file, ‘/’):

```
/Group1/Object1
```

New_Name

New string name for the object.

Keywords

SOFTLINK

If set the link will be a soft link. The default is to create a hard link.

NEW_LOC_ID

An integer giving the file or group identifier for the new link. This keyword is only used when linking to an object in a different file or group.

Example

See the example under H5F_CREATE.

Version History

6.2	Introduced

See Also

H5G_CREATE, H5G_UNLINK

H5G_MOVE

The H5G_MOVE procedure renames/moves an object within an HDF5 group or file.

Syntax

H5G_MOVE, *Loc_id*, *Src_Name*, *Dst_Name* [, NEW_LOC_ID=*value*]

Arguments

Loc_id

An integer giving the file or group identifier.

Src_Name

Original string name of the object.

Dst_Name

New string name for the object.

Keywords

NEW_LOC_ID

An integer giving the destination file or group identifier. This keyword is only used when linking to an object in a different file or group.

Example

See the example under H5F_CREATE.

Version History

6.2	Introduced

See Also

H5G_CREATE

H5G_OPEN

The H5G_OPEN function opens an existing group within an HDF5 file.

Syntax

Result = H5G_OPEN(*Loc_id*, *Name*)

Return Value

Returns the group's identifier number. This identifier can be released with the H5G_CLOSE.

Arguments

Loc_id

An integer representing the identifier of the file or group containing the group to be accessed.

Name

A string representing the name of the group to be accessed.

Keywords

None

Version History

5.6	Introduced

See Also

H5G_CLOSE

H5G_SET_COMMENT

Syntax | Arguments | Keywords | Example | Version History | See Also

The H5G_SET_COMMENT procedure sets a comment for a specified object.

Syntax

H5G_SET_COMMENT, *Loc_id*, *Name*, *Comment*

Arguments

Loc_id

An integer giving the file or group identifier containing the object.

Name

Name of the object within Loc_id whose comment is to be set or reset.

Comment

New comment for the object.

Keywords

None

Example

See the example under H5F_CREATE.

Version History

6.2	Introduced

See Also

H5G_CREATE

H5G_UNLINK

The H5G_UNLINK procedure removes the link to an object from a group.

Syntax

H5G_UNLINK, *Loc_id*, *Name*

Arguments

Loc_id

An integer giving the file or group identifier containing the object.

Name

Name of the object within Loc_id to unlink.

Keywords

None

Example

See the example under H5F_CREATE.

Version History

6.2	Introduced

See Also

H5G_CREATE, H5G_LINK

H5I_GET_FILE_ID

The H5I_GET_FILE_ID function retrieves an identifier for the file containing the specified object.

Syntax

Result = H5I_GET_FILE_ID(*Loc_id*)

Return Value

The *Result* is the identifier of the file.

Arguments

Loc_id

An integer giving the identifier of the object whose associated file identifier will be returned.

Keywords

NONE

Version History

6.2	Introduced

H5I_GET_TYPE

The H5I_GET_TYPE function returns the object’s type.

Syntax

Result = H5I_GET_TYPE(*Obj_id*)

Return Value

Returns a string representing the object type. Possible return values include:

- ‘FILE‘
- ‘GROUP‘
- ‘DATATYPE‘
- ‘DATASPACE‘
- ‘DATASET‘
- ‘ATTR‘
- ‘BADID‘

Arguments

Obj_id

An integer representing the object’s identifier for which to return the type.

Keywords

None

Version History

5.6	Introduced

H5R_CREATE

The H5R_CREATE function creates a reference to either an object or a dataspace region of a dataset.

Syntax

Result = H5R_CREATE(*Loc_id*, *Name* [,DATASPACE_ID=*value*])

Return Value

The *Result* is the reference pointing to the specified object. It is returned as either an integer, if an object reference is returned, or a named structure, if a dataspace region reference is returned.

Arguments

Loc_id

An integer giving the identifier used to locate the object being pointed to. This is the identifier of the object containing Name.

Name

The name of the object contained within Loc_id.

Keywords

DATASPACE_ID

An integer giving the identifier of the selection. Use of this keyword will create a dataspace region reference. If not supplied then an object reference will be created.

Example

See the example under H5F_CREATE.

Version History

6.2	Introduced

See Also

H5S_CREATE_SIMPLE

H5R_DEREFERENCE

The H5R_DEREFERENCE function opens a reference and returns the object identifier.

Syntax

Result = H5R_DEREFERENCE(*Loc_id*, *Reference*)

Return Value

The *Result* is the identifier number. This identifier should be released using the appropriate close procedure.

Arguments

Loc_id

An integer giving the identifier in which the reference dataset is located.

Reference

An integer or H5 reference structure giving the reference number to open.

Keywords

None

Version History

5.6	Introduced

See Also

H5R_GET_OBJECT_TYPE

H5R_GET_OBJECT_TYPE

The H5R_GET_OBJECT_TYPE function returns the type of object that an object reference points to.

Syntax

Result = H5R_GET_OBJECT_TYPE(*Loc_id*, *Reference*)

Return Value

The *Result* is a string giving the object type. Possible return values include:

- 'DATASET'
- 'GROUP'
- 'LINK'
- 'TYPE'
- 'UNKNOWN'

Arguments

Loc_id

An integer giving the identifier in which the reference dataset is located.

Reference

An integer giving the reference number to query.

Keywords

None

Version History

5.6	Introduced

See Also

H5R_DEREFERENCE

H5R_GET_REGION

The H5R_GET_REGION function retrieves a dataspace associated with a region reference.

Syntax

Result = H5R_GET_REGION(*Dataset_id*, *Reference*)

Return Value

The *Result* gives the identifier of the dataspace with the region selected.

Arguments

Dataset_id

An integer giving the identifier in which the reference dataset is located.

Reference

An H5 reference structure giving the reference number to open.

Keywords

None

Example

Assuming the file, 'hdf5_out.h5' was created using the example in H5F_CREATE, the dataspace region saved in the reference attached to the “Hanning” dataset could be obtained as follows:

```
fid = H5F_OPEN('hdf5_out.h5')
dataset_id = H5D_OPEN(fid,'Hanning')
attr_id = H5A_OPEN_NAME(dataset_id,'Ref')
ref = H5A_READ(attr_id)
dataspace_id = H5R_GET_REGION(dataset_id,ref)
```

Version History

6.2	Introduced

See Also

H5R_CREATE, H5D_CREATE, H5D_CLOSE

H5S_CLOSE

The H5S_CLOSE procedure releases and terminates access to a dataspace. After this routine is used, the dataspace’s identifier is no longer available.

Warning

Failure to release a dataspace using this procedure will result in resource leaks.

Syntax

H5S_CLOSE, *Dataspace_id*

Arguments

Dataspace_id

An integer representing the dataspace’s identifier to close.

Keywords

None

Version History

5.6	Introduced

See Also

H5D_GET_SPACE

H5S_COPY

The H5S_COPY function copies an existing dataspace.

Syntax

Result = H5S_COPY(*Dataspace_id*)

Return Value

Returns the dataspace's identifier number. The dataspace identifier can be released with the H5S_CLOSE.

Arguments

Dataspace_id

An integer representing the dataspace identifier to copy.

Keywords

None

Version History

5.6	Introduced

See Also

H5S_CREATE_SIMPLE, H5S_CLOSE

H5S_CREATE_SCALAR

Syntax | Return Value | Arguments | Keywords | Version History

The H5S_CREATE_SCALAR function creates a scalar dataspace.

Note

Scalar dataspaces have no dimensionality thus H5S_GET_SIMPLE_EXTENT_DIMS and H5S_GET_SIMPLE_EXTENT_NDIMS will both return 0.

Syntax

Result = H5S_CREATE_SCALAR()

Return Value

The *Result* gives the dataspace identifier. This identifier should be released with the H5S_CLOSE procedure.

Arguments

None

Keywords

None

Version History

6.2	Introduced

H5S_CREATE_SIMPLE

The H5S_CREATE_SIMPLE function creates a simple dataspace.

Syntax

Result = H5S_CREATE_SIMPLE(*Dimensions* [, MAX_DIMENSIONS=*vector*])

Return Value

Returns the dataspace's identifier number. This dataspace identifier can be released with the H5S_CLOSE.

Arguments

Dimensions

Set this argument to a vector containing the dimensions for the dataspace.

Note

The *Dimensions* argument should be specified in IDL's column-major order. Internally, the dimensions will be reversed to match HDF5/C's row-major order.

Keywords

MAX_DIMENSIONS

Set this keyword to a vector containing the maximum dimensions for the dataspace. The MAX_DIMENSIONS must have the same number of elements as the *Dimensions* argument. If MAX_DIMENSIONS is omitted then the maximum dimensions are set to *Dimensions*. You can use a value of -1 in MAX_DIMENSIONS to indicate an unlimited dimension.

Note

The values specified in the MAX_DIMENSIONS keyword should be equal to or greater than the corresponding values of the *Dimensions* argument.

Note

The MAX_DIMENSIONS keyword should be specified in IDL's column-major order. Internally, the dimensions will be reversed to match HDF5/C's row-major order.

Version History

5.6	Introduced

See Also

H5S_CLOSE, H5S_COPY

H5S_GET_SELECT_BOUNDS

The H5S_GET_SELECT_BOUNDS function retrieves the coordinates of the bounding box containing the current dataspace selection.

Syntax

Result = H5S_GET_SELECT_BOUNDS(*Dataspace_id*)

Return Value

Returns an (*m* x 2) array, where *m* is the number of dimensions (or rank) of the dataspace. The first row in the array is the starting coordinates of the bounding box, while the second row is the ending coordinates.

Arguments

Dataspace_id

An integer representing the dataspace's identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5S_GET_SIMPLE_EXTENT_NPOINTS, H5S_GET_SELECT_NPOINTS, H5S_GET_SELECT_ELEM_NPOINTS, H5S_GET_SELECT_HYPER_NBLOCKS

H5S_GET_SELECT_ELEM_NPOINTS

The H5S_GET_SELECT_ELEM_NPOINTS function determines the number of element points in the current dataspace selection.

Syntax

Result = H5S_GET_SELECT_ELEM_NPOINTS(*Dataspace_id*)

Return Value

Returns the number of element points.

Arguments

Dataspace_id

An integer representing the dataspace’s identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5S_GET_SELECT_BOUNDS, H5S_GET_SELECT_HYPER_NBLOCKS, H5S_GET_SELECT_NPOINTS, H5S_GET_SIMPLE_EXTENT_NPOINTS

H5S_GET_SELECT_ELEM_POINTLIST

The H5S_GET_SELECT_ELEM_POINTLIST function returns a list of the element points in the current dataspace selection.

Syntax

Result = H5S_GET_SELECT_ELEM_POINTLIST(*Dataspace_id* [, START=*value*] [, NUMBER=*value*])

Return Value

The *Result* is an (*m* x *n*) array, where *m* is the number of dimensions (or rank) of the dataspace, and *n* is the number of selected points. Each row contains the coordinates for an element selection point.

Arguments

Dataspace_id

An integer representing the dataspace's identifier to be queried.

Keywords

START

Set this keyword to an integer representing the point to start with, counting from 0. The default is START = 0.

NUMBER

Set this keyword to an integer representing the number of element points to return. The default is NUMBER = (N - START), where N is the total number of element points in the selection.

Version History

5.6	Introduced

See Also

H5S_GET_SELECT_ELEM_NPOINTS, H5S_GET_SELECT_NPOINTS

H5S_GET_SELECT_HYPER_BLOCKLIST

The H5S_GET_SELECT_HYPER_BLOCKLIST function returns a list of the hyperslab blocks in the current dataspace selection.

Syntax

Result = H5S_GET_SELECT_HYPER_BLOCKLIST(*Dataspace_id*
[, START=*value*] [, NUMBER=*value*])

Return Value

Returns an (*m* x 2*n*) array, where *m* is the number of dimensions (or rank) of the dataspace. The 2*n* rows of *Result* contain the list of blocks. The first row contains the start coordinates of the first block, followed by the next row which contains the opposite corner coordinates, followed by the next row which contains the start coordinates of the second block, etc.

Arguments

Dataspace_id

An integer representing the dataspace's identifier to be queried.

Keywords

START

Set this keyword to an integer representing the block to start with, counting from 0. The default is START = 0.

NUMBER

Set this keyword to an integer representing the number of blocks to return. The default is NUMBER = (N - START), where N is the total number of blocks in the selection.

Version History

5.6	Introduced

See Also

H5S_GET_SELECT_HYPER_NBLOCKS, H5S_GET_SELECT_NPOINTS

H5S_GET_SELECT_HYPER_NBLOCKS

The H5S_GET_SELECT_HYPER_NBLOCKS function determines the number of hyperslab blocks in the current dataspace selection.

Syntax

Result = H5S_GET_SELECT_HYPER_NBLOCKS(*Dataspace_id*)

Return Value

Returns the number of blocks.

Arguments

Dataspace_id

An integer representing the dataspace identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5S_GET_SELECT_BOUNDS, H5S_GET_SELECT_ELEM_NPOINTS, H5S_GET_SELECT_NPOINTS, H5S_GET_SIMPLE_EXTENT_NPOINTS

H5S_GET_SELECT_NPOINTS

The H5S_GET_SELECT_NPOINTS function determines the number of elements in a dataspace selection.

Syntax

Result = H5S_GET_SELECT_NPOINTS(*Dataspace_id*)

Return Value

Returns the number of elements.

Arguments

Dataspace_id

An integer representing the dataspace identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5S_GET_SELECT_BOUNDS, H5S_GET_SELECT_ELEM_NPOINTS, H5S_GET_SELECT_HYPER_NBLOCKS, H5S_GET_SIMPLE_EXTENT_NPOINTS

H5S_GET_SIMPLE_EXTENT_DIMS

The H5S_GET_SIMPLE_EXTENT_DIMS function returns the dimension sizes for a dataspace.

Syntax

Result = H5S_GET_SIMPLE_EXTENT_DIMS(*Dataspace_id*
 [, MAX_DIMENSIONS=*variable*])

Return Value

Returns a vector containing the dimension sizes.

Arguments

Dataspace_id

An integer representing the dataspace's identifier to be queried.

Keywords

MAX_DIMENSIONS

Set this keyword to a named variable to return the maximum dimension sizes for the dataspace.

Version History

5.6	Introduced

See Also

H5S_GET_SIMPLE_EXTENT_NDIMS, H5S_GET_SIMPLE_EXTENT_NPOINTS, H5S_GET_SIMPLE_EXTENT_TYPE

H5S_GET_SIMPLE_EXTENT_NDIMS

The H5S_GET_SIMPLE_EXTENT_NDIMS function determines the number of dimensions (or rank) of a dataspace.

Syntax

Result = H5S_GET_SIMPLE_EXTENT_NDIMS(*Dataspace_id*)

Return Value

Returns the number of dimensions.

Arguments

Dataspace_id

An integer representing the dataspace's identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5S_GET_SIMPLE_EXTENT_DIMS, H5S_GET_SIMPLE_EXTENT_NPOINTS, H5S_GET_SIMPLE_EXTENT_TYPE

H5S_GET_SIMPLE_EXTENT_NPOINTS

The H5S_GET_SIMPLE_EXTENT_NPOINTS function determines the number of elements in a dataspace.

Syntax

Result = H5S_GET_SIMPLE_EXTENT_NPOINTS(*Dataspace_id*)

Return Value

Returns the number of elements.

Arguments

Dataspace_id

An integer representing the dataspace's identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5S_GET_SIMPLE_EXTENT_DIMS, H5S_GET_SIMPLE_EXTENT_NDIMS, H5S_GET_SIMPLE_EXTENT_TYPE

H5S_GET_SIMPLE_EXTENT_TYPE

The H5S_GET_SIMPLE_EXTENT_TYPE function returns the current class of a dataspace.

Syntax

Result = H5S_GET_SIMPLE_EXTENT_TYPE(*Dataspace_id*)

Return Value

Returns a string containing the class. Possible values are:

- 'H5S_SCALAR'
- 'H5S_SIMPLE'
- 'H5S_COMPLEX'
- 'H5S_NO_CLASS'

Arguments

Dataspace_id

An integer representing the dataspace's identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5S_GET_SIMPLE_EXTENT_DIMS, H5S_GET_SIMPLE_EXTENT_NDIMS, H5S_GET_SIMPLE_EXTENT_NPOINTS

H5S_IS_SIMPLE

The H5S_IS_SIMPLE function determines whether a dataspace is a simple dataspace.

Syntax

Result = H5S_IS_SIMPLE(*Dataspace_id*)

Return Value

Returns 1 if the dataspace is simple and 0 if it is not.

Arguments

Dataspace_id

An integer representing the dataspace’s identifier to be queried.

Keywords

None

Version History

5.6	Introduced

H5S_OFFSET_SIMPLE

The H5S_OFFSET_SIMPLE procedure sets the selection offset for a simple dataspace. The offset allows the same shaped selection to be moved to different locations within the dataspace.

Syntax

H5S_OFFSET_SIMPLE, *Dataspace_id*, *Offset*

Arguments

Dataspace_id

An integer representing the dataspace's identifier on which to set the selection offset.

Offset

An *m*-element vector of integers, where *m* is the number of dataspace dimensions, containing the offsets.

Keywords

None

Version History

5.6	Introduced

See Also

H5S_GET_SELECT_BOUNDS, H5S_SELECT_ELEMENTS, H5S_SELECT_HYPERSLAB

H5S_SELECT_ALL

The H5S_SELECT_ALL procedure selects the entire extent of a dataspace.

Syntax

H5S_SELECT_ALL, *Dataspace_id*

Arguments

Dataspace_id

An integer representing the dataspace's identifier to be selected.

Keywords

None

Version History

5.6	Introduced

See Also

H5S_GET_SELECT_NPOINTS, H5S_SELECT_ELEMENTS, H5S_SELECT_HYPERSLAB, H5S_SELECT_NONE

H5S_SELECT_ELEMENTS

The H5S_SELECT_ELEMENTS procedure selects array elements to be included in the selection for a dataspace.

Syntax

H5S_SELECT_ELEMENTS, *Dataspace_id*, *Coordinates* [, /RESET]

Arguments

Dataspace_id

An integer representing the dataspace's identifier on which to set the selection.

Coordinates

An *m*-element vector, or an (*m* x *n*) array, where *m* is the number of dimensions (or rank) of the dataspace, and *n* is the number of selected points. Each row contains the coordinates for an element selection point.

Keywords

RESET

Set this keyword to replace the existing selection with the new *Coordinates*. The default is RESET = 0 which adds the new selection to the existing selection.

Note

The RESET keyword must be set (/RESET or RESET = 1) or the H5S_SELECT_ELEMENTS routine will result in an error message. This error message comes from the HDF5 library, which forces a default of RESET = 0 but insists on this keyword being set for this routine to work.

Version History

5.6	Introduced

See Also

H5S_GET_SELECT_ELEM_NPOINTS, H5S_GET_SELECT_ELEM_POINTLIST, H5S_GET_SELECT_NPOINTS, H5S_SELECT_HYPERSLAB

H5S_SELECT_HYPERSLAB

The H5S_SELECT_HYPERSLAB procedure selects a hyperslab region to be included in the selection for a dataspace.

Note

If all of the elements in the selected hyperslab region are already selected, then a new hyperslab region is not created.

Syntax

H5S_SELECT_HYPERSLAB, *Dataspace_id*, *Start*, *Count*, [, BLOCK=*vector*]
[, /RESET] [, STRIDE=*vector*]

Arguments

Dataspace_id

An integer representing the dataspace's identifier on which to set the selection.

Start

An *m*-element vector of integers, where *m* is the number of dataspace dimensions, containing the starting location for the hyperslab.

Count

An *m*-element vector of integers containing the number of blocks to select in each dimension.

Keywords

BLOCK

Set this keyword to an *m*-element vector of integers containing the size of a block. The default is a single element in each dimension (for example BLOCK is set to a vector of all 1's).

RESET

Set this keyword to replace the existing selection with the new selection. The default is RESET=0 which adds the new selection to the existing selection.

STRIDE

Set this keyword to an *m*-element vector of integers containing the number of elements to move in each dimension when selecting blocks. The default is to move a single element in each dimension (for example STRIDE is set to a vector of all 1's). STRIDE values must be greater than zero.

Version History

5.6	Introduced

See Also

H5S_GET_SELECT_HYPER_BLOCKLIST, H5S_GET_SELECT_HYPER_NBLOCKS, H5S_GET_SELECT_NPOINTS, H5S_SELECT_ELEMENTS

H5S_SELECT_NONE

The H5S_SELECT_NONE procedure resets the dataspace selection region to include no elements.

Syntax

H5S_SELECT_NONE, *Dataspace_id*

Arguments

Dataspace_id

An integer representing the dataspace's identifier to be reset.

Keywords

None

Version History

5.6	Introduced

See Also

H5S_GET_SELECT_NPOINTS, H5S_SELECT_ALL, H5S_SELECT_ELEMENTS, H5S_SELECT_HYPERSLAB

H5S_SELECT_VALID

The H5S_SELECT_VALID function verifies that the selection is within the extent of a dataspace.

Syntax

Result = H5S_SELECT_VALID(*Dataspace_id*)

Return Value

Returns 1 if the selection is within the dataspace and 0 if it is not.

Arguments

Dataspace_id

An integer representing the dataspace's identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5S_GET_SELECT_NPOINTS, H5S_SELECT_ELEMENTS, H5S_SELECT_HYPERSLAB

H5S_SET_EXTENT_NONE

Syntax | Arguments | Keywords | Version History | See Also

The H5S_SET_EXTENT_NONE removes the extent of a dataspace and sets the type to H5S_NO_CLASS. As such the dataspace cannot be resized or used in the creation of datasets or attributes.

Syntax

H5S_SET_EXTENT_NONE, *Dataspace_id*

Arguments

Dataspace_id

An integer giving the dataspace identifier.

Keywords

None

Version History

6.2	Introduced

See Also

H5S_CREATE_SIMPLE, H5S_SET_EXTENT_SIMPLE

H5S_SET_EXTENT_SIMPLE

Syntax | Arguments | Keywords | Version History | See Also

The H5S_SET_EXTENT_SIMPLE procedure sets or resets the extent of a dataspace.

Syntax

H5S_SET_EXTENT_SIMPLE, *Dataspace_id*, *Dimensions*
[,MAX_DIMENSIONS=*vector*]

Arguments

Dataspace_id

An integer giving the dataspace identifier.

Dimensions

An integer array or scalar giving the size of each array dimension. The number of elements in Dimensions defines the number of dimensions in the resulting array datatype.

Note
The values specified in the MAX_DIMENSIONS keyword should be equal to or greater than the corresponding values of the *Dimensions* argument.

Note
The Dimensions argument should be specified in IDL column-major order. Internally, the dimensions will be reversed to match HDF5/C row-major order.

Keywords

MAX_DIMENSIONS

A vector containing the maximum dimensions for the dataspace. MAX_DIMENSIONS must have the same number of elements as the Dimensions argument. If MAX_DIMENSIONS is omitted then the maximum dimensions are set to Dimensions. You can use a value of -1 in MAX_DIMENSIONS to indicate an unlimited dimension.

Version History

6.2	Introduced

See Also

H5S_CREATE_SIMPLE, H5S_SET_EXTENT_NONE

H5T_ARRAY_CREATE

The H5T_ARRAY_CREATE function creates an array datatype object.

Syntax

Result = H5T_ARRAY_CREATE(*Datatype_id*, *Dimensions*)

Return Value

The *Result* gives the identifier of the new datatype. The datatype identifier returned from this function should be released with H5T_CLOSE.

Arguments

Datatype_id

An integer giving the datatype identifier of the datatype of each element in the resulting array.

Dimensions

An integer array giving the size of each array dimension. The number of elements in Dimensions defines the number of dimensions in the resulting array datatype.

Note

The Dimensions argument should be specified in IDL column-major order. Internally, the dimensions will be reversed to match HDF5/C row-major order.

Keywords

None

Example

See the example under H5F_CREATE.

Version History

6.2	Introduced

See Also

H5T_IDL_CREATE, H5T_REFERENCE_CREATE

H5T_CLOSE

The H5T_CLOSE procedure releases the specified datatype’s identifier and releases resources used by it. After this routine is used, the datatype’s identifier is no longer available until the H5T_OPEN routine is used again to specify that datatype.

Syntax

H5T_CLOSE, *Datatype_id*

Arguments

Datatype_id

An integer representing the datatype’s identifier to be closed.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_OPEN

H5T_COMMIT

The H5T_COMMIT procedure commits a transient datatype to a file, creating a new named datatype.

Note

A named datatype can be shared by objects within the same HDF5 file, but not by objects in other files.

Syntax

H5T_COMMIT, *Loc_id*, *Name*, *Datatype_id*

Arguments

Loc_id

An integer giving the identifier of a file or group.

Name

The name of the new datatype.

Datatype_id

An integer giving the identifier of the datatype to commit.

Keywords

None

Example

See the example under H5F_CREATE.

Version History

6.2	Introduced

See Also

H5T_COMMITTED, H5T_IDL_CREATE

H5T_COMMITTED

The H5T_COMMITTED function determines whether a datatype is a named datatype or a transient type.

Syntax

Result = H5T_COMMITTED(*Datatype_id*)

Return Value

Returns 1 if the datatype is named and 0 if the datatype is transient.

Arguments

Datatype_id

An integer representing the datatyped identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_COMMIT

H5T_COMPOUND_CREATE

The H5T_COMPOUND_CREATE function creates a compound datatype object. See "Compound Datatypes" on page 124 for a discussion of compound datatypes.

Syntax

Result = H5T_COMPOUND_CREATE(*Datatype_id*, *Name*)

Return Value

Returns a long integer containing the identifier of the new compound datatype.

Note

Datatype identifiers created by this function should be released with H5T_CLOSE.

Arguments

Datatype_id

A long integer or array of long integers containing the datatype identifiers to use when creating the compound datatype. The number of elements in *Datatype_id* must match the number of elements in *Name*.

Name

A scalar string or string array containing the names to use when creating the compound datatype. The number of elements in *Name* must match the number of elements in *Datatype_id*.

Keywords

None

Example

```
dt1 = H5T_ENUM_CREATE()
H5T_ENUM_SET_DATA, dt1, ['Radar_1', 'Radar_2'], [1, 2]
dt2 = H5T_IDL_CREATE(1ul)
dt = H5T_COMPOUND_CREATE([dt1, dt2], ['Radar_Id', 'Ctrl_Num'])
```

Version History

6.3	Introduced

See Also

H5T_ARRAY_CREATE, H5T_IDL_CREATE, H5T_CLOSE

H5T_COPY

The H5T_COPY function copies an existing datatype. The returned type is transient and unlocked.

Syntax

Result = H5T_COPY(*Datatype_id*)

Return Value

Returns the datatype's identifier number. This identifier can be released with the H5T_CLOSE procedure.

Arguments

Datatype_id

An integer representing the datatype's identifier to be copied.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_CLOSE, H5T_OPEN

H5T_ENUM_CREATE

The H5T_ENUM_CREATE function creates an enumeration datatype object.

Note

Name/value pairs must be assigned to the datatype *before* it is used to create a dataset. The dataset stores the state of the datatype at the time the dataset is created; additional changes to the datatype will not be reflected in the dataset.

See "Enumeration Datatypes" on page 127 for a discussion of enumeration datatypes.

Syntax

Result = H5T_ENUM_CREATE()

Return Value

Returns a long integer containing the identifier of the new enumeration datatype.

Note

Datatype identifiers created by this function should be released with H5T_CLOSE.

Arguments

None

Keywords

None

Example

See "Enumeration Datatypes" on page 127 for an example using this routine.

Version History

6.3	Introduced

See Also

H5T_ENUM_INSERT, H5T_CLOSE

H5T_ENUM_GET_DATA

The H5T_ENUM_GET_DATA function retrieves the name/value pairs from an enumeration datatype object.

This routine is written in the IDL language. Its source code can be found in the file `h5t_enum_get_data.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = H5T_ENUM_GET_DATA(*Datatype_id*)

Return Value

Returns an array of `IDL_H5_ENUM` structures. Each structure has two fields:

Name	A string containing the name of the member
Value	The value of the corresponding member

Arguments

Datatype_id

A long integer containing the identifier of the enumeration datatype for which the data is desired.

Keywords

None

Version History

6.3	Introduced

See Also

H5T_ENUM_CREATE, H5T_ENUM_INSERT, H5T_ENUM_SET_DATA

H5T_ENUM_INSERT

The H5T_ENUM_INSERT procedure inserts a new member into an existing enumeration datatype.

Syntax

H5T_ENUM_INSERT, *Datatype_id*, *Name*, *Value*

Arguments

Datatype_id

A long integer containing the identifier of the enumeration datatype to which the new member will be added.

Name

A scalar string containing the name of the member to be added.

Value

A scalar integer giving the value of the member to be added.

Keywords

None

Version History

6.3	Introduced

See Also

H5T_ENUM_CREATE

H5T_ENUM_NAMEOF

The H5T_ENUM_NAMEOF function retrieves the name of a member of an enumeration datatype corresponding to the specified value.

Syntax

Result = H5T_ENUM_NAMEOF(*Datatype_id*, *Value*)

Return Value

Returns a string containing the name that corresponds to the specified value.

Arguments

Datatype_id

A long integer containing the identifier of the enumeration datatype.

Value

An integer containing the value of the datatype member for which the name should be returned.

Keywords

None

Version History

6.3	Introduced

See Also

H5T_ENUM_CREATE, H5T_ENUM_VALUEOF

H5T_ENUM_SET_DATA

The H5T_ENUM_SET_DATA procedure sets all the name/value pairs on an enumeration datatype object.

This routine is written in the IDL language. Its source code can be found in the file `h5t_enum_set_data.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

H5T_ENUM_SET_DATA, *Datatype_id*, *Data*, *Values*

Arguments

Datatype_id

A long integer containing the identifier of the enumeration datatype.

Data

A string array or array of named structures:

- If *Data* is a string array, it contains the *names* of the members of the enumeration datatype, and the *Values* argument is required.
- If *Data* is an array of named structures, the *Values* argument is ignored. The structures that make up *Data* must be of type IDL_H5_ENUM, which has the following signature:

```
{IDL_H5_ENUM, NAME:'', VALUE:0}
```

Values

An integer array containing the *values* of the members of the enumeration datatype. The *Values* array must have the same number of elements as the string array specified by *Data*. This argument is ignored if *Data* is an array of structures.

Keywords

None

Version History

6.3	Introduced

See Also

H5T_ENUM_CREATE, H5T_ENUM_GET_DATA, H5T_ENUM_INSERT

H5T_ENUM_VALUEOF

The H5T_ENUM_VALUEOF function retrieves the value of a member of an enumeration datatype corresponding to the specified name.

Syntax

Result = H5T_ENUM_VALUEOF(*Datatype_id*, *Name*)

Return Value

Returns an integer containing the value that corresponds to the specified name.

Arguments

Datatype_id

A long integer containing the identifier of the enumeration datatype.

Name

A string containing the name of the datatype member for which the value should be returned.

Keywords

None

Version History

6.3	Introduced

See Also

H5T_ENUM_CREATE, H5T_ENUM_NAMEOF

H5T_ENUM_VALUES_TO_NAMES

The H5T_ENUM_VALUES_TO_NAMES function converts values to the corresponding names of an enumeration datatype.

This routine is written in the IDL language. Its source code can be found in the file `h5t_enum_values_to_names.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = H5T_ENUM_VALUES_TO_NAMES(*Datatype_id*, *Values*)

Return Value

Returns a string array with the same number of elements as *Values*. Each element in *Result* will be the name corresponding to the value in *Values*. If an individual value specified in *Values* is not present in the enumeration datatype, the corresponding element in *Result* will be a null string.

Arguments

Datatype_id

A long integer containing the identifier of the enumeration datatype.

Values

An integer array of values for which names are desired.

Note

The H5D_READ function returns an array suitable for use as the *Values* argument.

Keywords

None

Version History

6.3	Introduced

See Also

H5T_ENUM_NAMEOF

H5T_EQUAL

The H5T_EQUAL function determines whether two datatype identifiers refer to the same datatype.

Syntax

Result = H5T_EQUAL(*Datatype_id1*, *Datatype_id2*)

Return Value

Returns 1 if the identifiers refer to the same datatype and 0 if they do not.

Arguments

Datatype_id1

An integer representing the first datatype identifier.

Datatype_id2

An integer representing the second datatype identifier.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_COPY

H5T_GET_ARRAY_DIMS

The H5T_GET_ARRAY_DIMS function returns the dimension sizes for an array datatype object.

Syntax

Result = H5T_GET_ARRAY_DIMS(*Datatype_id* [, PERMUTATIONS=*variable*])

Return Value

Returns a vector containing the dimension sizes.

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Keywords

PERMUTATIONS

Set this keyword to a named variable in which to return the dimension permutations (C versus FORTRAN).

Version History

5.6	Introduced

See Also

H5T_GET_ARRAY_NDIMS

H5T_GET_ARRAY_NDIMS

The H5T_GET_ARRAY_NDIMS function determines the number of dimensions (or rank) of an array datatype object.

Syntax

Result = H5T_GET_ARRAY_NDIMS(*Datatype_id*)

Return Value

Returns the number of dimensions.

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_GET_ARRAY_DIMS

H5T_GET_CLASS

The H5T_GET_CLASS function returns the datatype’s class.

Syntax

Result = H5T_GET_CLASS(*Datatype_id*)

Return Value

Returns a string containing the datatype’s class. Possible return values include:

- ‘H5T_INTEGER’
- ‘H5T_FLOAT’
- ‘H5T_TIME’
- ‘H5T_STRING’
- ‘H5T_BITFIELD’
- ‘H5T_OPAQUE’
- ‘H5T_COMPOUND’
- ‘H5T_REFERENCE’
- ‘H5T_ENUM’
- ‘H5T_VLEN’
- ‘H5T_ARRAY’
- ‘H5T_NO_CLASS’

Arguments

Datatype_id

An integer representing the datatype’s identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_GET_SIZE, H5T_GET_SUPER

H5T_GET_CSET

The H5T_GET_CSET function returns the character set type of a string datatype.

Syntax

Result = H5T_GET_CSET(*Datatype_id*)

Return Value

Returns a string containing the character set type. Possible values are:

- 'ASCII' — US ASCII
- 'ERROR'

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Keywords

None

Version History

5.6	Introduced

H5T_GET_EBIAS

The H5T_GET_EBIAS function returns the exponent bias of a floating-point type.

Syntax

Result = H5T_GET_EBIAS(*Datatype_id*)

Return Value

Returns the exponent bias.

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_GET_FIELDS

H5T_GET_FIELDS

The H5T_GET_FIELDS function retrieves information about the positions and sizes of bit fields within a floating-point datatype.

Syntax

Result = H5T_GET_FIELDS(*Datatype_id*)

Return Value

Returns a structure named H5T_GET_FIELDS containing the following tags:

TYPE_ID

The datatype’s identifier *Datatype_id*.

SIGN_POS

The position of the floating-point sign bit.

EXP_POS

The bit position of the exponent.

EXP_SIZE

The size of the exponent in bits.

MAN_POS

The bit position of the mantissa.

MAN_SIZE

The size of the mantissa in bits.

Arguments

Datatype_id

An integer representing the datatype’s identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_GET_EBIAS, H5T_GET_INPAD, H5T_GET_NORM, H5T_GET_OFFSET, H5T_GET_ORDER, H5T_GET_PAD, H5T_GET_PRECISION

H5T_GET_INPAD

The H5T_GET_INPAD function returns the padding method for unused internal bits within a floating-point datatype.

Syntax

Result = H5T_GET_INPAD(*Datatype_id*)

Return Value

Returns the padding method. Possible values are:

- 0 — Background set to zeroes
- 1 — Background set to ones
- 2 — Background left unchanged

Arguments

Datatype_id

An integer representing the datatype identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_GET_FIELDS

H5T_GET_MEMBER_CLASS

The H5T_GET_MEMBER_CLASS function returns the datatype class of a compound datatype member.

Syntax

Result = H5T_GET_MEMBER_CLASS(*Datatype_id*, *Member*)

Return Value

Returns a string containing the datatype class. Possible values are:

- ‘H5T_INTEGER’
- ‘H5T_FLOAT’
- ‘H5T_TIME’
- ‘H5T_STRING’
- ‘H5T_BITFIELD’
- ‘H5T_OPAQUE’
- ‘H5T_COMPOUND’
- ‘H5T_REFERENCE’
- ‘H5T_ENUM’
- ‘H5T_VLEN’
- ‘H5T_ARRAY’
- ‘H5T_NO_CLASS’

Arguments

Datatype_id

An integer representing the datatype’s identifier to be queried.

Member

An integer representing the member index, starting at zero.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_GET_MEMBER_NAME, H5T_GET_MEMBER_OFFSET, H5T_GET_MEMBER_TYPE, H5T_GET_NMEMBERS

H5T_GET_MEMBER_INDEX

The H5T_GET_MEMBER_INDEX function retrieves the index of a specified member of a compound or enumeration datatype.

Syntax

Result = H5T_GET_MEMBER_INDEX(*Datatype_id*, *Field_name*)

Return Value

Returns an integer containing the index of the specified datatype member.

Arguments

Datatype_id

A long integer containing the identifier of the compound or enumeration datatype.

Field_name

A string containing the name of the field or element for which the index will be returned.

Keywords

None

Version History

6.3	Introduced

See Also

H5T_ENUM_CREATE, H5T_GET_MEMBER_VALUE

H5T_GET_MEMBER_NAME

The H5T_GET_MEMBER_NAME function retrieves the name of a compound or enumeration datatype member.

Syntax

Result = H5T_GET_MEMBER_NAME(*Datatype_id*, *Member*)

Return Value

Returns a string containing the name of the specified datatype member.

Arguments

Datatype_id

A long integer containing the identifier of the compound or enumeration datatype.

Member

An integer containing the member index. Member indices are zero-based.

Keywords

None

Version History

5.6	Introduced
6.3	Added support for enumeration datatypes

See Also

H5T_GET_MEMBER_CLASS, H5T_GET_MEMBER_OFFSET, H5T_GET_MEMBER_TYPE, H5T_GET_NMEMBERS

H5T_GET_MEMBER_OFFSET

The H5T_GET_MEMBER_OFFSET function returns the byte offset of a field within a compound datatype.

Syntax

Result = H5T_GET_MEMBER_OFFSET(*Datatype_id*, *Member*)

Return Value

Returns an integer representing the byte offset.

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Member

An integer representing the member index, starting at zero.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_GET_MEMBER_CLASS, H5T_GET_MEMBER_NAME, H5T_GET_MEMBER_TYPE, H5T_GET_NMEMBERS

H5T_GET_MEMBER_TYPE

The H5T_GET_MEMBER_TYPE function returns the datatype identifier for a specified member within a compound datatype.

Syntax

Result = H5T_GET_MEMBER_TYPE(*Datatype_id*, *Member*)

Return Value

Returns the datatype identifier. This identifier should be closed using H5T_CLOSE.

Arguments

Datatype_id

An integer representing the datatype’s identifier to be queried.

Member

An integer representing the member index, starting at zero.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_GET_MEMBER_CLASS, H5T_GET_MEMBER_NAME, H5T_GET_MEMBER_OFFSET, H5T_CLOSE, H5T_GET_NMEMBERS

H5T_GET_MEMBER_VALUE

The H5T_GET_MEMBER_VALUE function retrieves the value of an enumeration datatype member.

Syntax

Result = H5T_GET_MEMBER_VALUE(*Datatype_id*, *Member*)

Return Value

Returns an integer containing the value of the specified datatype member.

Arguments

Datatype_id

A long integer containing the identifier of the enumeration datatype.

Member

An integer containing the member index. Member indices are zero-based.

Keywords

None

Version History

6.3	Introduced

See Also

H5T_ENUM_CREATE, H5T_GET_MEMBER_INDEX

H5T_GET_NMEMBERS

The H5T_GET_NMEMBERS function returns the number of fields in a compound datatype.

Syntax

Result = H5T_GET_NMEMBERS(*Datatype_id*)

Return Value

Returns the number of fields.

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_GET_MEMBER_CLASS, H5T_GET_MEMBER_NAME, H5T_GET_MEMBER_OFFSET, H5T_GET_MEMBER_TYPE

H5T_GET_NORM

The H5T_GET_NORM function returns the mantissa normalization of a floating-point datatype.

Syntax

Result = H5T_GET_NORM(*Datatype_id*)

Return Value

Returns a string containing the mantissa normalization. Possible values are:

- 'IMPLIED' — Most-significant bit of mantissa not stored, always 1
- 'MSBSET' — Most-significant bit of mantissa is always 1
- 'NORM' — Mantissa is not normalized
- 'ERROR'

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_GET_FIELDS

H5T_GET_OFFSET

The H5T_GET_OFFSET function returns the bit offset of the first significant bit in an atomic datatype. The offset is the number of bits of padding that follows the significant bits (for big endian) or precedes the significant bits (for little endian).

Syntax

Result = H5T_GET_OFFSET(*Datatype_id*)

Return Value

Returns the bit offset.

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_GET_FIELDS

H5T_GET_ORDER

The H5T_GET_ORDER function returns the byte order of an atomic datatype.

Syntax

Result = H5T_GET_ORDER(*Datatype_id*)

Return Value

Returns a string representing the byte order. Possible values are:

- 'LE' — Little endian
- 'BE' — Big endian
- 'VAX' — VAX mixed ordering
- 'NONE'
- 'ERROR'

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_GET_INPAD, H5T_GET_PAD, H5T_GET_PRECISION

H5T_GET_PAD

The H5T_GET_PAD function returns the padding method of the least significant bit (*lsb*) and most significant bit (*msb*) of an atomic datatype.

Syntax

Result = H5T_GET_PAD(*Datatype_id*)

Return Value

Returns a two-element vector [*lsb*, *msb*]. Possible values are:

- 0 — Background set to zeroes
- 1 — Background set to ones
- 2 — Background left unchanged.

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_GET_INPAD, H5T_GET_ORDER, H5T_GET_PRECISION

H5T_GET_PRECISION

The H5T_GET_PRECISION function returns the precision in bits of an atomic datatype. The precision is the number of significant bits which, unless padded, is 8 times larger than the byte size from H5T_GET_CSET.

Syntax

Result = H5T_GET_PRECISION(*Datatype_id*)

Return Value

Returns the bit precision, or 0 if the datatype is not atomic.

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_GET_INPAD, H5T_GET_ORDER, H5T_GET_PAD, H5T_GET_SIZE

H5T_GET_SIGN

The H5T_GET_SIGN function returns the sign type for an integer datatype.

Syntax

Result = H5T_GET_SIGN(*Datatype_id*)

Return Value

Returns the sign type. Possible values are:

- -1 — Error
- 0 — Unsigned integer type
- 1 — Two's complement signed integer type

Arguments

Datatype_id

An integer representing the datatype’s identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_GET_ORDER, H5T_GET_PAD, H5T_GET_PRECISION

H5T_GET_SIZE

The H5T_GET_SIZE function returns the size of a datatype in bytes.

Syntax

Result = H5T_GET_SIZE(*Datatype_id*)

Return Value

Represents the size (in bytes) of the first element found within the datatype.

Note

When H5T_GET_SIZE is given a datatype containing a string, it will return the number of characters + 1.

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_GET_CLASS, H5T_GET_SUPER

H5T_GET_STRPAD

The H5T_GET_STRPAD function returns the padding method for a string datatype.

Syntax

Result = H5T_GET_STRPAD(*Datatype_id*)

Return Value

Returns a string containing the padding method. Possible values are:

- 'NULLTERM' — Null terminate (like C)
- 'NULLPAD' — Pad with zeroes
- 'SPACEPAD' — Pad with spaces (like FORTRAN)
- 'ERROR'

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_GET_CSET, H5T_GET_SIZE

H5T_GET_SUPER

The H5T_GET_SUPER function returns the base datatype from which a datatype is derived.

Syntax

Result = H5T_GET_SUPER(*Datatype_id*)

Return Value

Returns the base datatype's identifier number. This identifier can be released with the H5T_CLOSE.

Arguments

Datatype_id

An integer representing the datatype's identifier to be queried.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_GET_CLASS, H5T_GET_SIZE

H5T_GET_TAG

The H5T_GET_TAG function retrieves a tag string from an opaque data type. See "Opaque Datatypes" on page 125 for a discussion of opaque datatypes.

Syntax

Result = H5T_GET_TAG(*Datatype_id*)

Return Value

Returns a string containing the tag associated with the specified opaque datatype, or a null string if no tag exists.

Arguments

Datatype_id

A long integer containing the identifier of the opaque datatype.

Keywords

None

Version History

6.3	Introduced

See Also

H5T_IDL_CREATE, H5T_SET_TAG

H5T_IDL_CREATE

The H5T_IDL_CREATE function creates a datatype object based on the IDL type of the supplied data.

Syntax

Result = H5T_IDL_CREATE(*Data* [,MEMBER_NAMES=*vector*] [, /OPAQUE])

Return Value

Returns a long integer containing the identifier of the new datatype.

Note

Datatype identifiers created by this function should be released with H5T_CLOSE.

Arguments

Data

An IDL variable containing the type of data that will be used by the resulting datatype. If a structure is passed in a compound datatype will be created based on the fields of the structure. The following table shows how IDL data types are converted to HDF5 datatypes. Pointers, complex numbers, and object references cannot be written to HDF5 datatypes.

IDL type	HDF5 Datatype
Byte	H5T_NATIVE_UINT8
Integer	H5T_NATIVE_INT16
Unsigned integer	H5T_NATIVE_UINT16
Long integer	H5T_NATIVE_INT32
Unsigned long integer	H5T_NATIVE_UINT32
64-bit Integer	H5T_NATIVE_INT64
Unsigned 64-bit integer	H5T_NATIVE_UINT64

Table 3-15: IDL Types and Corresponding HDF5 Datatypes

IDL type	HDF5 Datatype
Floating point	H5T_NATIVE_FLOAT
Double-precision floating	H5T_NATIVE_DOUBLE
String	H5T_C_S1
Structure	(Member datatypes)

Table 3-15: IDL Types and Corresponding HDF5 Datatypes (Continued)

Note

If the data is an array, the datatype is constructed from the first element in the array. If an HDF5 array datatype is desired, then the datatype returned in this routine should be passed into H5T_ARRAY_CREATE. Using the first element could affect the size of a string datatype. All elements of a string datatype will have the same length, or number of characters. Strings smaller than the datatype length will be stored appropriately, but strings longer than the datatype length will be truncated. The size of the returned datatype will include a null termination character and thus will be one more than the number of characters in the string. For example:

```
datatype_id = H5T_IDL_CREATE('dog')
```

This produces a datatype with a size of 4. A dataset created with this datatype will only store up to 4 characters per element of the data being written. The following:

```
datatype_id = H5T_IDL_CREATE(['dog', 'dragon'])
```

will still produce a datatype with a size of 4 because the first element of the array is used when creating the datatype. When creating a string datatype the longest string needed should be used. Note that an excessively long string could result in a bloated file.

Keywords

MEMBER_NAMES

A string vector giving the name of each member of the compound datatype. This keyword is ignored if *Data* is not an IDL structure. If *Data* is an IDL structure and this keyword is not provided, the member names will be constructed from the field names in the structure, converting all letters to uppercase and all non-alphanumeric characters to underscores. If the number of elements in MEMBER_NAMES is less than the number of elements in the structure, field names will be used for member names where needed. If the number of elements in MEMBER_NAMES is greater than the number of elements in the structure, the extra string values will be ignored.

Elements of MEMBER_NAMES are assigned to fields in a depth-first, left-to-right traversal of the structure. For example, if *Data* contains a structure that looks like:

```
{ a:0l, b:{d:0l, e:0l}, c:0l }
```

and MEMBER_NAMES contains:

```
['cat', 'dog', 'dragon', 'emu']
```

then the resulting compound datatype uses the name `'cat'` to refer to the datatype created from field `a`, `'dog'` to refer to field `b`, `'dragon'` to refer to field `d`, and `'emu'` to refer to field `e`. Since only four names are provided, the compound datatype uses the name `'C'` to refer to field `c`.

OPAQUE

Set this keyword to create an opaque datatype. The size of the datatype will be determined by the size of the *Data* argument.

Note

Opaque datatypes store data as a simple string of bytes, regardless of the form the data is in when written to the datatype. An opaque datatype is always a single element, regardless of the number of bytes. See "Opaque Datatypes" on page 125 for additional information.

Example

See the example under H5F_CREATE.

Version History

6.2	Introduced
6.3	Added support for opaque datatypes and OPAQUE keyword

See Also

H5T_ARRAY_CREATE, H5T_CLOSE, H5T_REFERENCE_CREATE

H5T_IDLTYPE

The H5T_IDLTYPE function returns the IDL type code corresponding to a datatype.

Note

This function is not part of the standard HDF5 interface, but is provided as a programming convenience.

Syntax

Result = H5T_IDLTYPE(*Datatype_id*
[, ARRAY_DIMENSIONS=*variable*][, STRUCTURE=*variable*])

Return Value

The *Result* gives the IDL type code.

Note

For a list of IDL type codes and their definitions, see "IDL Type Codes and Names" in the *IDL Reference Guide* manual under the SIZE function.

Arguments

Datatype_id

An integer giving the datatype identifier for which to return the IDL type code.

Keywords

ARRAY_DIMENSIONS

Set this keyword to a named variable in which to return a vector containing the array dimensions, if the datatype is an array. If the datatype is not an array, then a scalar value of 0 is returned.

STRUCTURE

Set this keyword to a named variable in which to return the IDL structure definition, if the datatype is a compound datatype. If the datatype is not compound, then a scalar value of 0 is returned.

Version History

5.6	Introduced

See Also

H5T_MEMTYPE

H5T_INSERT

The H5T_INSERT procedure adds a new member to the end of a compound datatype.

Syntax

H5T_INSERT, *Datatype_id*, *Name*, *Field_id*

Arguments

Datatype_id

An integer giving the identifier of the compound datatype to modify.

Name

Name of the field to insert.

Field_id

An integer giving the identifier of the datatype of the field to insert.

Keywords

None

Example

See the example under H5F_CREATE.

Version History

6.2	Introduced

See Also

H5T_IDL_CREATE

H5T_MEMTYPE

The H5T_MEMTYPE function returns the native memory datatype corresponding to a file datatype.

Note

This function is not part of the standard HDF5 interface, but is provided as a programming convenience.

Syntax

Result = H5T_MEMTYPE(*Datatype_id*)

Return Value

The *Result* gives the datatype identifier. If the file datatype is not immutable, then the memory datatype identifier should be closed using H5T_CLOSE.

Note

For a list of IDL type codes and their definitions, see "IDL Type Codes and Names" in the *IDL Reference Guide* manual under the SIZE function.

Arguments

Datatype_id

An integer giving the file datatype identifier for which to return the memory datatype.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_IDLTYPE

H5T_OPEN

The H5T_OPEN function opens a named datatype.

Syntax

Result = H5T_OPEN(*Loc_id*, *Name*)

Return Value

Returns the datatype’s identifier number. This identifier can be released with the H5T_CLOSE.

Arguments

Loc_id

An integer representing the identifier of the file or group containing the datatype.

Name

A string representing the name of the datatype to be accessed.

Keywords

None

Version History

5.6	Introduced

See Also

H5T_CLOSE

H5T_REFERENCE_CREATE

The H5T_REFERENCE_CREATE function creates a reference datatype object.

Syntax

Result = H5T_REFERENCE_CREATE([/REGION])

Return Value

The *Result* is either an integer (if an object reference is created) or a structure (if a dataspace region reference is created) giving the identifier of the new datatype. The datatype identifier returned from this function should be released with H5T_CLOSE.

Arguments

None

Keywords

REGION

If set a dataspace region reference will be created. The default is to create an object reference.

Example

See the example under H5F_CREATE.

Version History

6.2	Introduced

See Also

H5T_ARRAY_CREATE, H5T_IDL_CREATE, H5T_CLOSE

H5T_SET_TAG

The H5T_SET_TAG procedure sets a tag string on an opaque data type. See "Opaque Datatypes" on page 125 for a discussion of opaque datatypes.

Syntax

H5T_SET_TAG(*Datatype_id*, *Tag*)

Arguments

Datatype_id

A long integer containing the identifier of the opaque datatype.

Tag

A string containing the new tag for the opaque datatype.

Note

Tag strings can contain a maximum of 255 characters.

Keywords

None

Version History

6.3	Introduced

See Also

H5T_GET_TAG, H5T_IDL_CREATE

H5T_STR_TO_VLEN

The H5T_STR_TO_VLEN function converts an IDL string array to an IDL_H5_VLEN array of strings.

This routine is written in the IDL language. Its source code can be found in the file `h5t_str_to_vlen.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = H5T_STR_TO_VLEN(*Array* [, /NO_COPY])

Return Value

Returns an array of named structures of type IDL_H5_VLEN, each with a single field PDATA containing a pointer to a scalar string.

Arguments

Array

A string array to be converted to an HDF5 variable length array.

Keywords

NO_COPY

Set this keyword to delete the contents of *Array* after creating the corresponding array of IDL_H5_VLEN structures. The default is to leave a copy of the data in *Array*.

Version History

6.3	Introduced

See Also

H5T_VLEN_TO_STR, H5T_VLEN_CREATE

H5T_VLEN_CREATE

The H5T_VLEN_CREATE function creates a variable length array datatype object. See “Variable Length Array Datatypes” on page 129 for a discussion of variable length array datatypes.

Syntax

Result = H5T_VLEN_CREATE(*Datatype_id*)

Return Value

Returns a long integer containing the identifier of the new variable length array datatype.

Note

Datatype identifiers created by this function should be released with H5T_CLOSE.

Arguments

Datatype_id

A long integer containing the datatype identifier to use when creating the variable length array datatype. All elements of the array will be of this datatype.

Keywords

None

Example

See “Variable Length Array Datatypes” on page 129 for an example using this routine.

Version History

6.3	Introduced

See Also

H5T_ARRAY_CREATE, H5T_IDL_CREATE, H5T_CLOSE

H5T_VLEN_TO_STR

The H5T_VLEN_TO_STR function converts an IDL_H5_VLEN array of strings to an IDL string array.

This routine is written in the IDL language. Its source code can be found in the file `h5t_vlen_to_str.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = H5T_VLEN_TO_STR(*Array* [, /PTR_FREE])

Return Value

Returns an IDL string array.

Arguments

Array

An array of named structures of type IDL_H5_VLEN, each with a PDATA field pointing to a scalar string.

Keywords

PTR_FREE

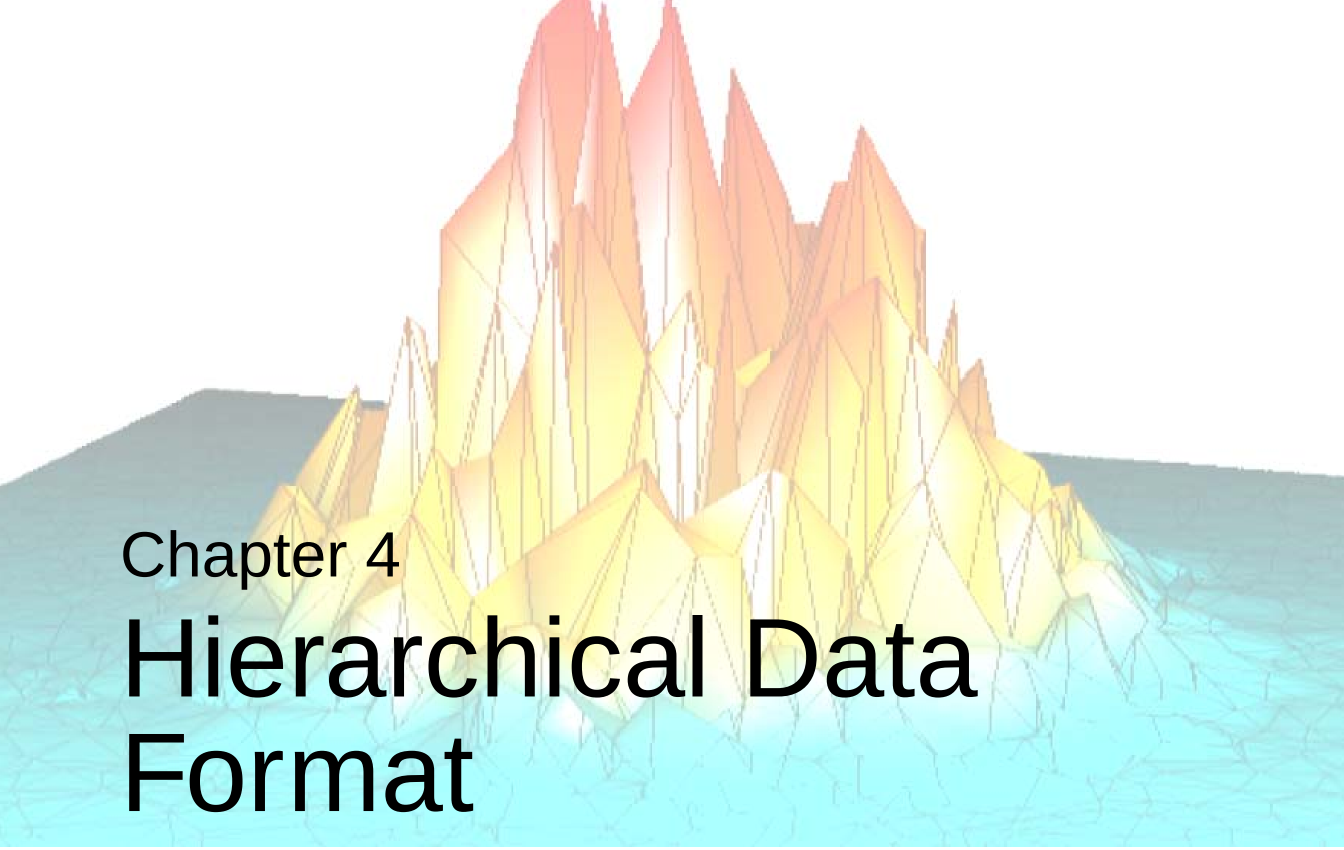
Set this keyword to free the pointers in the input array of IDL_H5_VLEN structures. The default is to not free the pointers.

Version History

6.3	Introduced

See Also

H5T_STR_TO_VLEN, H5T_VLEN_CREATE



Chapter 4
Hierarchical Data Format

The following topics are covered in this appendix:

Overview of the HDF Format

The Hierarchical Data Format (HDF) is a multi-object file format that facilitates the transfer of various types of data between machines and operating systems. HDF is a product of the National Center for Supercomputing Applications (NCSA). HDF is designed to be flexible, portable, self-describing and easily extensible for future enhancements or compatibility with other standard formats. The HDF library contains interfaces for storing and retrieving images and multi-dimensional scientific data. This version of IDL supports HDF 4.1r5.

Note

On the AIX platform, the HDF library supports version 4.1r3.

IDL's HDF routines all begin with the prefix "HDF_".

Further information about HDF can be found on the World Wide Web at the HDF Information Server:

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

Alternately, you can send e-mail to:

`hdfhelp@ncsa.uiuc.edu.`

HDF Interfaces

There are two basic interfaces to HDF files: the single-file application interface and the multiple-file application interface. These interfaces support eight different types (or “models”) of data access. The table below lists the different models and the names of the IDL routines that access those models. Each model is described in more detail after the table.

Model	IDL Routine Name Prefix
24-bit raster	HDF_DF24_
annotation data	HDF_DFAN_
palette data	HDF_DFP_
8-bit raster	HDF_DFR8_
scientific data	HDF_SD_
multi-file scientific data	HDF_SD_
VData	HDF_VD_
VGroup	HDF_VG_

Table 4-1: Data Access Models and Routine Prefixes

Single File Application Interfaces

In this mode, access is limited to one file at a time. This interface supports the 8-bit raster, 24-bit raster, palette, scientific data, and annotation models. The interfaces are described in more detail after the table.

- 8-bit Raster Model: The HDF_DFR8_ routines access 8-bit images.
- Palette Model: The HDF_DFP_ routines are used to work with the HDF_DFR8_ routines to manipulate palettes associated with 8-bit HDF images.
- 24-bit Raster Model: The HDF_DFR24_ routines access 24-bit images.
- Scientific Data Models (SDs): Used to manipulate arrays of arbitrary dimension and type. Under this model, an array accompanied by a record of its data type, dimensions and descriptors is called a Scientific Dataset (SD).

- Annotation Model: The annotation model is used to describe the contents of the file through such items as labels, data descriptors, and dimension scales.
- Vdata Model: This interface allows for the creation of customized tables. Each table consists of a series of Vdata records whose values are stored in fixed length fields. As described in more detail in the Vdata example below, a Vdata can contain three kinds of identifying information: a Vdata name, Vdata Class, and multiple Vdata field names. The Vdata model is accessed through the routines that begin with the HDF_VD_ prefix.
- Vgroup Model: A collection of one or more data objects, Vdata sets, or Vgroups is known as a Vgroup. Each Vgroup can be given a Vgroup name and Vgroup class. The Vgroup model is accessed through the routines that begin with the HDF_VG_ prefix.

Multi-File Application Interface

The HDF_SD_ routines allow operations on more than one file at a time. This multi-file interoperability is achieved through HDF's use of a modified version of the NetCDF library. IDL's interface to HDF's multi-file capability is the HDF_SD_SETEXTFILE routine.

Creating HDF Files

The following IDL commands should be used to create a new HDF file:

- HDF_OPEN: Call this procedure first to open an HDF file. The CREATE keyword must be set if you want to create a new file instead of opening an existing one.
- HDF_DFAN_ADDFDS: Optionally, add a file description.
- HDF_DFAN_ADDFID: Optionally, add a file annotation.

Adding Data to an HDF File

The routines used to add data to an HDF file vary based on the interface model being used:

- To add an 8-bit image (with or without a palette), use HDF_DFR8_ADDIMAGE or DFR8_PUTIMAGE.
- To add a palette, use HDF_DFP_ADDPAL or HDF_DFP_PUTPAL.
- To add a 24-bit image, use HDF_DF24_ADDIMAGE or HDF_DF24_PUTIMAGE.
- To add a Multi-File Scientific Dataset, use the following commands:
  - HDF_SD_CREATE or HDF_SD_SELECT to create an SDS or select an existing one.
  - HDF_SD_DIMSET to set dimension information.
  - HDF_SD_ATTRSET to set attribute information.
  - HDF_SD_SETINFO to insert optional information about the data.
  - HDF_SD_ADDDATA to insert the data.
  - HDF_SD_SETEXTFILE to move the data to an external file (optional).
  - HDF_SD_ENDACCESS to end access to the SDS.
- To add a Vdata, use the following commands:
  - HDF_VD_ATTACH to get a Vdata identifier.
  - HDF_VD_SETINFO to write information about the Vdata (optional).
  - HDF_VD_FDEFINE to prepare non-trivial fields (optional).
  - HDF_VD_WRITE to write the Vdata.

- To add a Vdata to a Vgroup, use the following commands:
  - HDF_VG_ATTACH to get a Vgroup identifier.
  - HDF_VG_SETINFO to set the Vgroup name and class (optional).
  - HDF_VG_INSERT to add the Vdata to a Vgroup.
  - HDF_VG_DETACH to close the Vgroup.
  - HDF_CLOSE to close the file.

HDF Examples

Example Code

Two example files that demonstrate the use of the HDF routines can be found in the `examples/doc/sdf` subdirectory of the IDL distribution. The file `hdf_info.pro` prints a summary of basic information about an HDF file. The file `hdf_rdwr.pro` creates a new HDF file and then reads the information back from that file.

HDF Scientific Dataset ID Numbers

IDL's HDF_SD_ routines can accept two different types of ID numbers. Documentation for these routines in the *IDL Reference Guide* refers to these ID numbers as the SDinterface_id and SDdataset_id arguments.

The SDinterface_id is the Scientific Dataset interface ID. There is only one SDinterface_id per HDF file. For each actual dataset used, you will also need an SDdataset_id, which is the ID for the particular dataset.

Some routines, such as HDF_SD_ATTRFIND, accept either an SDinterface_id or an SDdataset_id. In these cases, the documentation refers to the ID as an SD_id, meaning that either type of ID is accepted.

IDL and HDF Data Types

HDF and IDL support many different data types. Many of the HDF routines allow you to perform a data type conversion "on the fly" by setting keywords such as FLOAT. When the data type desired is not explicitly specified, IDL uses the conversions shown in the following tables. Note that single-precision floating-point is the default data type and that the complex data type is not supported.

When writing IDL data to an HDF file, IDL data types are converted to the HDF data types shown in the following table:

IDL Data Type	HDF Data Type
BYTE	DFNT_UINT8 (IDL bytes are unsigned)
INT	DFNT_INT16
UINT	DFNT_UINT16
LONG	DFNT_INT32
ULONG	DFNT_UINT32
FLOAT	DFNT_FLOAT32
DOUBLE	DFNT_DOUBLE
STRING	DFNT_CHAR8

Table 4-2: Type Conversions when Writing IDL Data to an HDF File

When reading data from an HDF file, HDF data types are converted to the IDL data types shown in the following table:

HDF Data Type	IDL Data Type
DFNT_CHAR8	STRING
DFNT_UINT8	BYTE
DFNT_INT16	INT
DFNT_UINT16	UINT
DFNT_INT32	LONG
DFNT_UINT32	ULONG
DFNT_INT64	LONG
DFNT_UINT64	ULONG
DFNT_FLOAT32 or DFNT_NONE	FLOAT

Table 4-3: Type Conversions when Reading HDF Data into IDL

HDF type codes for the supported HDF data types are shown in the table below:

HDF Data Type	Type Code
DFNT_UCHAR	3
DFNT_CHAR	4
DFNT_FLOAT32	5
DFNT_FLOAT64	6
DFNT_INT8	20
DFNT_UINT8	21
DFNT_INT16	22
DFNT_UINT16	23
DFNT_INT32	24

Table 4-4: HDF Data Type Codes

HDF Data Type	Type Code
DFNT_UINT32	25
DFNT_INT64	26
DFNT_UINT64	27

Table 4-4: HDF Data Type Codes (Continued)

Common HDF Tag Numbers

The following table lists common HDF tag numbers and their meanings.

Tag Number	Meaning
030	Version Identifier
100	File Identifier
101	File Description
102	Tag Identifier
103	Tag Description
104	Data Identifier Label
105	Data Identifier Annotation
106	Number Type
107	Machine Type
200	Obsolete
201	Obsolete
202	Obsolete
203	Obsolete
204	Obsolete
300	RIG Image Dimension
301	Raster Image Look Up Table (LUT)

Table 4-5: Common HDF Tag Numbers

Tag Number	Meaning
302	Raster Image
303	Compressed Raster Image
306	Raster Image Group (RIG)
307	RIG LUT Dimension
308	RIG Matte Dimension
309	Raster Image Matte Data
310	Raster Image Color Correction
311	Raster Image Color Format
312	Raster Image Aspect Ratio
400	Composite Image Descriptor
500	XY Position
602	Vector Image - Tek4014 Stream
603	Vector Image - Tek4105 Stream
701	SD Dimension Record
702	SD Data
703	SD Scales
704	SD Labels
705	SD Units
706	SD Formats
707	SD Max/Min
708	SD Coordinates
710	SD Link
720	SD Descriptor (NDG)
731	SD Calibration Information

Table 4-5: Common HDF Tag Numbers (Continued)

Tag Number	Meaning
732	SD Fill Value
1962	Vdata Description
1963	Vdata
1965	Vgroup

Table 4-5: Common HDF Tag Numbers (Continued)

Alphabetical Listing of HDF Routines

The HDF routines are listed in the following section.

Note

The routines HDF_BROWSER and HDF_READ, introduced in IDL version 5.1, allow you to read HDF data files and import data into IDL using a graphical user interface. Using these two routines, you can avoid the need to use most of the rest of IDL's HDF interface. HDF_BROWSER and HDF_READ are discussed in the *IDL Reference Guide*.

HDF_AN_ANNLEN

HDF_AN_ANNLIST

HDF_AN_ATYPE2TAG

HDF_AN_CREATE

HDF_AN_CREATEF

HDF_AN_END

HDF_AN_ENDACCESS

HDF_AN_FILEINFO

HDF_AN_GET_TAGREF

HDF_AN_ID2TAGREF

HDF_AN_NUMANN

HDF_AN_READANN

HDF_AN_SELECT

HDF_AN_START

HDF_AN_TAG2ATYPE

HDF_AN_TAGREF2ID

HDF_AN_WRITEANN

HDF_BROWSER

HDF_CLOSE

HDF_DELDD

HDF_DF24_ADDIMAGE

HDF_DF24_GETIMAGE

HDF_DF24_GETINFO

HDF_DF24_LASTREF

HDF_DF24_NIMAGES

HDF_DF24_READREF

HDF_DF24_RESTART

HDF_DFAN_ADDFDS

HDF_DFAN_ADDFID

HDF_DFAN_GETDESC

HDF_DFAN_GETFDS

HDF_DFAN_GETFID

HDF_DFAN_GETLABEL

HDF_DFAN_LABLIST

HDF_DFAN_LASTREF

HDF_DFAN_PUTDESC

HDF_DFAN_PUTLABEL

HDF_DFP_ADDPAL

HDF_DFP_GETPAL

HDF_DFP_LASTREF

HDF_DFP_NPALS

HDF_DFP_PUTPAL

HDF_DFP_READREF

HDF_DFP_RESTART

HDF_DFP_WRITEREF

HDF_DFR8_ADDIMAGE

HDF_DFR8_GETIMAGE

HDF_DFR8_GETINFO

HDF_DFR8_LASTREF

HDF_DFR8_NIMAGES

HDF_DFR8_PUTIMAGE

HDF_DFR8_READREF

HDF_DFR8_RESTART

HDF_DFR8_SETPALETTE

HDF_DUPDD

HDF_EXISTS

HDF_GR_ATTRINFO

HDF_GR_CREATE

HDF_GR_END

HDF_GR_ENDACCESS

HDF_GR_FILEINFO

HDF_GR_FINDATTR

HDF_GR_GETATTR

HDF_GR_GETCHUNKINFO

HDF_GR_GETIMINFO

HDF_GR_GETLUTID

HDF_GR_GETLUTINFO

HDF_GR_IDTOREF

HDF_GR_LUTTOREF

HDF_GR_NAMETOINDEX

HDF_GR_READIMAGE

HDF_GR_READLUT

HDF_GR_REFTOINDEX

HDF_GR_SELECT

HDF_GR_SETATTR

HDF_GR_SETCHUNK

HDF_GR_SETCHUNKCACHE

HDF_GR_SETCOMPRESS

HDF_GR_SETEXTERNALFILE

HDF_GR_START

HDF_GR_WRITEIMAGE

HDF_GR_WRITELUT

HDF_HDF2IDLTYPE

HDF_IDL2HDFTYPE

HDF_ISHDF

HDF_LIB_INFO

HDF_NEWREF

HDF_NUMBER

HDF_OPEN

HDF_PACKDATA

HDF_READ

HDF_SD_ADDDATA

HDF_SD_ATTRFIND

HDF_SD_ATTRINFO

HDF_SD_ATTRSET

HDF_SD_CREATE

HDF_SD_DIMGET

HDF_SD_DIMGETID

HDF_SD_DIMSET

HDF_SD_END

HDF_SD_ENDACCESS

HDF_SD_FILEINFO

HDF_SD_GETDATA

HDF_SD_GETINFO

HDF_SD_IDTOREF

HDF_SD_ISCOORDVAR

HDF_SD_NAMETOINDEX

HDF_SD_REFTOINDEX

HDF_SD_SELECT

HDF_SD_SETCOMPRESS

HDF_SD_SETEXTFILE

HDF_SD_SETINFO

HDF_SD_START

HDF_UNPACKDATA

HDF_VD_ATTACH

HDF_VD_ATTRFIND

HDF_VD_ATTRINFO

HDF_VD_ATTRSET

HDF_VD_DETACH

HDF_VD_FDEFINE

HDF_VD_FEXIST

HDF_VD_FIND

HDF_VD_GET

HDF_VD_GETID

HDF_VD_GETINFO

HDF_VD_INSERT

HDF_VD_ISATTR

HDF_VD_ISVD

HDF_VD_ISVG

HDF_VD_LONE

HDF_VD_NATTRS

HDF_VD_READ

HDF_VD_SEEK

HDF_VD_SETINFO

HDF_VD_WRITE

HDF_VG_ADDTR

HDF_VG_ATTACH

HDF_VG_DETACH

HDF_VG_GETID

HDF_VG_GETINFO

HDF_VG_GETNEXT

HDF_VG_GETTR

HDF_VG_GETTRS

HDF_VG_INQTR

HDF_VG_INSERT

HDF_VG_ISVD

HDF_VG_ISVG

HDF_VG_LONE

HDF_VG_NUMBER

HDF_VG_SETINFO

HDF_AN_ANNLEN

This function returns the number of characters contained in the HDF AN annotation specified by the annotation identifier *Annotation_id*.

Syntax

Result = HDF_AN_ANNLEN(*Annotation_id*)

Return Value

The number of characters contained in the HDF AN annotation.

Arguments

Annotation_id

Annotation identifier returned by HDF_AN_CREATE, HDF_AN_CREATEF, or HDF_AN_SELECT.

Keywords

None

Version History

5.2	Introduced

HDF_AN_ANNLIST

This function obtains a list of identifiers of the annotations that are of the type specified by the parameter Annotation_type and are attached to the object identified by its tag, *Object_tag*, and its reference number, *Object_ref*.

Syntax

Result = HDF_AN_ANNLIST(*Annotation_id*, *Annotation_type*, *Object_tag*, *Object_ref*, *Annotation_list*)

Return Value

Returns SUCCEED (0) or FAIL (-1) otherwise.

Arguments

Annotation_id

HDF AN interface identifier returned by HDF_AN_START.

Annotation_type

Type of the annotation. Since this routine is implemented only to obtain the identifiers of data annotations and not file annotations, the valid values of *Annotation_type* are:

- 0 = data label
- 1 = data description

Object_tag

HDF tag of the object.

Object_ref

HDF reference number of the object.

Annotation_list

A named variable that will contain the annotation identifiers.

Keywords

None

Version History

5.2	Introduced

HDF_AN_ATYPE2TAG

This function returns the HDF tag that corresponds to the annotation type specified by the parameter *Annotation_type*.

Syntax

Result = HDF_AN_ATYPE2TAG(*Annotation_type*)

Return Value

Returns the HDF annotation tag (*Annotation_tag*) if successful, or not found (0) otherwise.

Arguments

Annotation_type

Type of the annotation. The following table lists the valid values of *Annotation_type* in the left column and the corresponding values for the returned annotation tag on the right.

Annotation Type	HDF Annotation Tag
0 = Data Label (AN_DATA_LABEL)	104 (DFTAG_DIL)
1= Data Description (AN_DATA_DESC)	105 (DFTAG_DIA)
2 = File Label (AN_FILE_LABEL)	100 (DFTAG_FID)
3 = File Description (AN_FILE_DESC)	101 (DFTAG_FD)

Table 4-6: Valid Annotation_type and Annotation_tag values.

Keywords

None

Version History

5.2	Introduced

HDF_AN_CREATE

This function creates an HDF AN data annotation of type *Annotation_type* for the object specified by its HDF tag, *Object_tag*, and its HDF reference number, *Object_ref*. Use HDF_AN_CREATEF to create a file annotation. Currently, the user must write to a newly-created annotation before creating another annotation of the same type. Creating two consecutive annotations of the same type causes the second call to HDF_AN_CREATE to return FAIL (-1).

Syntax

Result = HDF_AN_CREATE(*Annotation_id*, *Object_tag*, *Object_ref*, *Annotation_type*)

Return Value

Returns the data annotation identifier (*Annotation_id*) if successful or FAIL (-1) otherwise.

Arguments

Annotation_id

HDF_AN_ INTERFACE identifier returned by HDF_AN_START.

Object_tag

HDF tag of the object to be annotated.

Object_ref

HDF reference number of the object to be annotated.

Annotation_type

Type of the data annotation.

The returned data annotation identifier can represent either a data label or a data description. Valid values for Annotation_type are:

- 0 = data label
- 1 = data description

Keywords

None

Version History

5.2	Introduced

HDF_AN_CREATEF

This function creates an HDF AN file annotation of the type specified by the parameter *Annotation_type*. Use HDF_AN_CREATE to create a data annotation. Currently, the user must write to a newly-created annotation before creating another annotation of the same type. Creating two consecutive annotations of the same type causes the second call to HDF_AN_CREATE to return FAIL (-1)

Syntax

Result = HDF_AN_CREATEF(*Annotation_id*, *Annotation_type*)

Return Value

Returns the file annotation identifier (*Annotation_id*) if successful or FAIL (-1) otherwise.

Arguments

Annotation_id

HDF_AN_ INTERFACE identifier returned by HDF_AN_START.

Annotation_type

Type of the file annotation. The file annotation identifier returned can either represent a file label or a file description. Valid values for *Annotation_type* are:

- 2 = file label
- 3 = file description

Keywords

None

Version History

5.2	Introduced

HDF_AN_END

This procedure terminates access to the HDF AN interface identified by *Annotation_id*, which is previously initialized by a call to HDF_AN_START. Note that there must be one call to HDF_AN_END for each call to HDF_AN_START.

Syntax

HDF_AN_END, *Annotation_id*

Arguments

Annotation_id

HDF AN interface identifier returned by HDF_AN_START.

Keywords

None

Version History

5.2	Introduced

HDF_AN_ENDACCESS

This procedure terminates access to the annotation identified by the parameter Annotation_id. Note that there must be one call to HDF_AN_ENDACCESS for every call to HDF_AN_SELECT, HDF_AN_CREATE or HDF_AN_CREATEF.

Syntax

HDF_AN_ENDACCESS, *Annotation_id*

Arguments

Annotation_id

Annotation identifier returned by HDF_AN_CREATE, HDF_AN_CREATEF or HDF_AN_SELECT.

Keywords

None

Version History

5.2	Introduced

HDF_AN_FILEINFO

This function retrieves the total number of the four kinds of annotations and stores them in the appropriate parameters. Note that the numbers of data labels and descriptions refer to the total number of data labels and data descriptions in the file, not for a specific object. Use HDF_AN_NUMANN to determine these numbers for a specific object. This function is generally used to find the range of acceptable indices for HDF_AN_SELECT calls.

Syntax

Result = HDF_AN_FILEINFO(*Annotation_id*, *n_file_labels*, *n_file_descs*,
n_data_labels, *n_data_descs*)

Return Value

Returns SUCCEED (0) if successful or FAIL (-1) otherwise.

Arguments

Annotation_id

HDF AN interface identifier returned by HDF_AN_START.

n_file_labels

A named variable that will contain the number of file labels.

n_file_descs

A named variable that will contain the number of file descriptions.

n_data_labels

A named variable that will contain the total number of data labels of all data objects in the file.

n_data_descs

A named variable that will contain the total number of data descriptions of all data objects in the file.

Keywords

None

Version History

5.2	Introduced

HDF_AN_GET_TAGREF

This function retrieves the HDF tag and reference number of the annotation identified by its index and by its annotation type.

Syntax

Result = HDF_AN_GET_TAGREF(*Annotation_id*, *index*, *Annotation_type*, *Annotation_tag*, *Annotation_ref*)

Return Value

Returns SUCCEED (0) if successful or FAIL (-1) otherwise.

Arguments

Annotation_id

HDF AN interface identifier returned by HDF_AN_START.

Index

Index of the annotation. This parameter is a nonnegative integer and is less than the total number of annotations of type *Annotation_type* in the file. Use HDF_AN_FILEINFO to obtain the total number of annotations of each type in the file.

Annotation_type

Type of the annotation. The following table lists the valid values of the parameter Annotation_type in the left column, and the corresponding values of the parameter Annotation_tag in the right column.

Annotation Type	HDF Annotation Tag
0 = Data Label (AN_DATA_LABEL)	104 (DFTAG_DIL)
1= Data Description (AN_DATA_DESC)	105 (DFTAG_DIA)

Table 4-7: Valid Annotation_type and Annotation_tag values.

Annotation Type	HDF Annotation Tag
2 = File Label (AN_FILE_LABEL)	100 (DFTAG_FID)
3 = File Description (AN_FILE_DESC)	101 (DFTAG_FD)

Table 4-7: Valid Annotation_type and Annotation_tag values.

Annotation_tag

A named variable that will contain the HDF tag of the annotation.

Annotation_ref

A named variable that will contain the HDF reference number of the annotation.

Keywords

None

Version History

5.2	Introduced

HDF_AN_ID2TAGREF

This function retrieves the HDF tag/reference number pair of the HDF AN annotation identified by its annotation identifier.

Syntax

Result = HDF_AN_ID2TAGREF(*Annotation_id*, *Annotation_tag*, *Annotation_ref*)

Return Value

Returns SUCCEED (0) if successful or FAIL (-1) otherwise.

Arguments

Annotation_id

HDF AN annotation identifier returned by HDF_AN_SELECT, HDF_AN_CREATE or HDF_AN_CREATEF.

Annotation_tag

A named variable that will contain the HDF tag of the annotation. Possible values returned in Annotation_tag are:

- 104 = data label (DFTAG_DIL)
- 105 = data description (DFTAG_DIA)
- 100 = file label (DFTAG_FID)
- 101 = file description (DFTAG_FD)

Annotation_ref

A named variable that will contain the HDF reference number of the annotation.

Keywords

None

Version History

5.2	Introduced

HDF_AN_NUMANN

This function returns the total number of HDF AN annotations that are of a given type and that are attached to the object identified by its HDF tag and its HDF reference number.

Syntax

Result = HDF_AN_NUMANN(*Annotation_id*, *Annotation_type*, *Object_tag*, *Object_ref*)

Return Value

Returns the number of annotations or FAIL (-1) otherwise.

Arguments

Annotation_id

HDF AN interface identifier returned by HDF_AN_START.

Annotation_type

Type of the annotation. The following table lists the valid values of the parameter Annotation_type in the left column, and the corresponding values of the parameter Annotation_tag in the right column.

Annotation Type	HDF Annotation Tag
0 = Data Label (AN_DATA_LABEL)	104 (DFTAG_DIL)
1 = Data Description (AN_DATA_DESC)	105 (DFTAG_DIA)
2 = File Label (AN_FILE_LABEL)	100 (DFTAG_FID)
3 = File Description (AN_FILE_DESC)	101 (DFTAG_FD)

Table 4-8: Valid Annotation_type and Annotation_tag values.

Object_tag

HDF tag of the object.

Object_ref

HDF reference number of the object. Since this routine is implemented only to obtain the total number of data annotations and not file annotations, the valid values of Annotation_type are:

- 0 = data label
- 1 = data description

To obtain the total number of file annotations or all data annotations, use HDF_AN_FILEINFO.

Keywords

None

Version History

5.2	Introduced

HDF_AN_READANN

This function reads the HDF AN annotation identified by the annotation identifier and stores the annotation into a variable.

Syntax

Result = HDF_AN_READANN(*Annotation_id*, *annotation*
[, LENGTH=*characters*])

Return Value

Returns SUCCEED (0) if successful or FAIL (-1) otherwise.

Arguments

Annotation_id

Annotation identifier returned by HDF_AN_CREATE, HDF_AN_CREATEF or HDF_AN_SELECT.

Annotation

A named variable that will contain the annotation.

Keywords

LENGTH

Specifies the number of characters to be read from the annotation argument. If LENGTH is not set, or LENGTH is greater than the number of characters in annotation, then the entire annotation is read.

Version History

5.2	Introduced

HDF_AN_SELECT

This function obtains the HDF AN identifier of the annotation specified by its index and by its annotation type.

Syntax

Result = HDF_AN_SELECT(*Annotation_id*, *index*, *Annotation_type*)

Return Value

Returns the annotation identifier (*Annotation_id*) if successful or FAIL (-1) otherwise.

Arguments

Annotation_id

HDF_AN_ INTERFACE identifier returned by HDF_AN_START.

Index

Location of the annotation in the file. This parameter is a nonnegative integer and is less than the total number of annotations of type Annotation_type in the file minus 1. Use HDF_AN_FILEINFO to obtain the total number of annotations of each type in the file.

Annotation_type

Type of the annotation. Valid values of Annotation_type are:

- 0 = data labels
- 1 = data descriptions
- 2 = file labels
- 3 = file descriptions

Keywords

None

Version History

5.2	Introduced

HDF_AN_START

This function initializes the HDF AN interface for the specified file. A call to HDF_AN_START is required before any HDF AN functions can be invoked. HDF_AN_START is used with the HDF_AN_END function to define the extent of an HDF AN session. A call to HDF_AN_END is required for each call to HDF_AN_START.

Syntax

Result = HDF_AN_START(*file_id*)

Return Value

Returns the HDF AN interface identifier (Annotation_id) if successful or FAIL (-1) otherwise.

Arguments

File_id

File identifier returned by HDF_OPEN. Note that each call to HDF_OPEN *must* be terminated with a call to HDF_CLOSE.

Keywords

None

Version History

5.2	Introduced

HDF_AN_TAG2ATYPE

This function returns the HDF AN annotation type that corresponds to the specified HDF annotation tag.

Syntax

Result = HDF_AN_TAG2ATYPE(*Annotation_tag*)

Return Value

Returns the annotation type if successful or FAIL (-1) otherwise.

Arguments

Annotation_tag

HDF tag of the annotation.

The following table lists the valid values of Annotation_tag in the left column and the corresponding values of the returned annotation type in the right column.

Annotation Type	HDF Annotation Tag
0 = Data Label (AN_DATA_LABEL)	104 (DFTAG_DIL)
1= Data Description (AN_DATA_DESC)	105 (DFTAG_DIA)
2 = File Label (AN_FILE_LABEL)	100 (DFTAG_FID)
3 = File Description (AN_FILE_DESC)	101 (DFTAG_FD)

Table 4-9: Valid Annotation_type and Annotation_tag values.

Keywords

None

Version History

5.2	Introduced

HDF_AN_TAGREF2ID

This function returns the HDF AN identifier of the annotation specified by its HDF tag and its HDF reference number.

Syntax

Result = HDF_AN_TAGREF2ID(*Annotation_id*, *Annotation_tag*, *Annotation_ref*)

Return Value

Returns the annotation identifier (*Annotation_id*) if successful or FAIL (-1) otherwise.

Arguments

Annotation_id

HDF_AN_ INTERFACE identifier returned by HDF_AN_START.

Annotation_tag

HDF tag of the annotation. Valid values are:

- 104 = data label (DFTAG_DIL)
- 105 = data description (DFTAG_DIA)
- 100 = file label (DFTAG_FID)
- 101 = file description (DFTAG_FD)

Annotation_ref

HDF reference number of the annotation.

Keywords

None

Version History

5.2	Introduced

HDF_AN_WRITEANN

This function writes the annotation text provided in the parameter *annotation* to the HDF AN annotation specified by the parameter *Annotation_id*.

Syntax

Result = HDF_AN_WRITEANN(*Annotation_id*, *annotation*
[, LENGTH=*characters*])

Return Value

Returns SUCCEED (0) if successful or FAIL (-1) otherwise.

Arguments

Annotation_id

Annotation identifier returned by HDF_AN_CREATE, HDF_AN_CREATEF, or HDF_AN_SELECT.

Annotation

Text or IDL variable to be written as the annotation.

Keywords

LENGTH

Length of the annotation text to be written. If not specified, the entire annotation will be written. If the keyword LENGTH is set, then only LENGTH characters of the annotation will be written. If the annotation has already been written, HDF_AN_WRITEANN will overwrite the current text.

Version History

5.2	Introduced

HDF_BROWSER

See HDF_BROWSER in the *IDL Reference Guide*.

HDF_CLOSE

The HDF_CLOSE procedure closes the HDF file associated with the given file handle.

Syntax

HDF_CLOSE, *FileHandle*

Arguments

FileHandle

The HDF file handle returned from a previous call to HDF_OPEN.

Keywords

None

Version History

Pre 4.0	Introduced

See Also

HDF_OPEN

HDF_DELDD

The HDF_DELDD procedure deletes a tag or reference from the list of data descriptors in an HDF file.

Syntax

HDF_DELDD, *FileHandle*, *Tag*, *Ref*

Arguments

FileHandle

The HDF file handle returned from a previous call to HDF_OPEN.

Tag

The data descriptor tag to delete.

Reference

The data descriptor reference number to delete.

Keywords

None

Version History

Pre 4.0	Introduced

HDF_DF24_ADDIMAGE

The HDF_DF24_ADDIMAGE procedure writes a 24-bit raster image to an HDF file. The interlace is set automatically based upon the dimensions of the image being written: ARR(3, Width, Height) for pixel interlace, ARR(Width, 3, Height) for scan-line interlace, and ARR(Width, Height, 3) for scan-plane interlace.

Note

HDF_DF24_ADDIMAGE chooses an interlace based upon the location of the '3'-sized dimension. For 3x3xN, 3xNx3 and Nx3x3 images, if the first '3' encountered is supposed to be a width or height, HDF_DF24_ADDIMAGE will choose the 'wrong' interlace. However, as long as one reads in the image using the same interlace, the image will be read correctly anyway. Avoid writing 24-bit-deep raster images with a width or height of 3 pixels.

Note

Input data is converted to bytes before being written to the file, as images in the DF24 HDF model are necessarily byte images.

Syntax

HDF_DF24_ADDIMAGE, *Filename*, *Image* [, /FORCE_BASELINE{useful only if QUALITY<25}] [, /JPEG | , /RLE] [, QUALITY=*value*{0 to 100}]

Arguments

Filename

A scalar string containing the name of the file to be written.

Image

A 3-dimensional array of values representing the 3 planes (Red, Green, and Blue) of the 24-bit image. One of the dimensions *must* be 3 (e.g., a 3 x 100 x 100 array).

Keywords

FORCE_BASELINE

Set this keyword to force the JPEG quantization tables to be constrained to the range 1 to 255. This provides full baseline compatibility with external JPEG applications, but only makes a difference if the QUALITY keyword is set to a value less than 25. The default is TRUE.

JPEG

Set this keyword to compress the image being added using the JPEG (Joint Photographic Expert Group) method. Note that JPEG compression is *lossy*; see WRITE_JPEG in the *IDL Reference Guide* for more information about when this method is appropriate. (In other words, using JPEG compression to reduce the size of an image changes the values of the pixels and hence may alter the meaning of the corresponding data.) Setting either the QUALITY or the FORCE_BASELINE keyword implies this method.

QUALITY

Set this keyword equal to the JPEG "quality" desired. This value should be in the range 0 (terrible image quality but excellent compression) to 100 (excellent image quality but minimum compression). The default is 75. Setting this keyword implies that the JPEG keyword is set. Lower values of QUALITY produce higher compression ratios and smaller files.

RLE

Set this keyword to store the image using run length compression. RLE compression is lossless, and is recommended for images where data retention is critical.

Version History

4.0	Introduced

See Also

WRITE_JPEG

HDF_DF24_GETIMAGE

The HDF_DF24_GETIMAGE procedure reads a 24-bit raster image from an HDF file. The default is to use the same format for reading as that used in writing the image. Note: it is slower to read an image in a different interlace than the one in which the image was originally written.

Syntax

HDF_DF24_GETIMAGE, *Filename*, *Image* [, /LINE | , /PIXEL | , /PLANE]

Arguments

Filename

A scalar string containing the name of the file to be read.

Image

A named variable in which the image data is returned.

Keywords

LINE

Set this keyword to force the image to be read with scan-line interlace.

PIXEL

Set this keyword to force the image to be read with pixel interlace.

PLANE

Set this keyword to force the image to be read with scan-plane interlace.

Version History

4.0	Introduced

HDF_DF24_GETINFO

The HDF_DF24_GETINFO procedure retrieves information about the current 24-bit HDF image.

Syntax

HDF_DF24_GETINFO, *Filename*, *Width*, *Height*, *Interlace*

Arguments

Filename

A string containing the name of the file to be read.

Width

A named variable in which the width of the image is returned.

Height

A named variable in which the height of the image is returned.

Interlace

A named variable in which the interface method is returned. The returned value is 0 for pixel interlacing, 1 for scan-line interlacing, and 2 for scan-plane interlacing.

Keywords

None

Examples

```
; Open the file myhdf.hdf:
h = HDF_OPEN('myhdf.hdf')
; Return information about the 24-bit image:
HDF_DF24_GETINFO, 'myhdf.hdf', width, height, interlace
; Print information about the returned variables:
HELP, width, height, interlace
HDF_CLOSE('myhdf.hdf') ; Close the HDF file.
```

IDL Output

If the image were 536 by 412 pixels, and scan-line interlaced, IDL would print:

```
WIDTH LONG = 536
HEIGHT LONG = 412
INTERLACE LONG = 1
```

Example Code

For a more detailed example, see the file `hdf_info.pro`, located in the `examples/doc/sdf` subdirectory of the IDL distribution.

Version History

4.0	Introduced

See Also

HDF_DF24_GETIMAGE, HDF_DF24_LASTREF, HDF_DF24_NIMAGES, HDF_DF24_READREF, HDF_DF24_RESTART

HDF_DF24_LASTREF

The HDF_DF24_LASTREF function returns the reference number of the most recently read or written 24-bit image in an HDF file.

Syntax

Result = HDF_DF24_LASTREF( )

Return Value

Returns the reference number of the most recently read or written image.

Arguments

None

Keywords

None

Examples

```
; Open an HDF file.
h=HDF_OPEN('myhdf.hdf')
PRINT, HDF_DF24_LASTREF()
; IDL prints 0, meaning that the call was successful,
; but no reference number was available.

; Create a 3D array, representing a 24-bit image:
a = BINDGEN(3,100,100)

; Write the 24-bit image to the file:
HDF_DF24_ADDIMAGE, 'myhdf.hdf', a

PRINT, HDF_DF24_LASTREF()
; IDL prints a reference number for the last operation
; (for example, 2). Note the reference number is not
; simply a 1-based "image number"; the reference number
; could easily be "2" or "3", etc.

; Write another image to the file:
HDF_DF24_ADDIMAGE, 'myhdf.hdf', a
```

```
; Print the last reference number:
PRINT, HDF_DF24_LASTREF()
PRINT, HDF_DF24_NIMAGES('myhdf.hdf')
; IDL prints "2" because we’ve written two images to the file.
; Close the file
HDF_CLOSE, h
```

Version History

4.0	Introduced

See Also

HDF_DF24_ADDIMAGE, HDF_DF24_GETIMAGE, HDF_DF24_GETINFO, HDF_DF24_NIMAGES, HDF_DF24_READREF, HDF_DF24_RESTART, HDF_DFR8_LASTREF

HDF_DF24_NIMAGES

The HDF_DF24_NIMAGES function returns the number of 24-bit images in an HDF file.

Syntax

Result = HDF_DF24_NIMAGES(*Filename*)

Return Value

Returns the number of images in the file or -1 if the specified file is invalid or damaged

Arguments

Filename

A string containing the name of the file to be searched.

Keywords

None

Examples

```
; Open HDF file:
h = HDF_OPEN('myhdf.hdf')
; Return the number of 24-bit images in the file:
number = HDF_DF24_NIMAGES('myhdf.hdf')
; Print information about the returned value. If there were five
; images in the file, IDL would print NUMBER LONG = 5
HELP, number
; Close the HDF file:
HDF_CLOSE, h
```

Version History

4.0	Introduced

See Also

HDF_DF24_GETIMAGE, HDF_DF24_GETINFO, HDF_DF24_READREF, HDF_DF24_RESTART, HDF_DFR8_NIMAGES

HDF_DF24_READREF

The HDF_DF24_READREF procedure sets the reference number of the image in an HDF file to be read by the next call to HDF_DF24_GETIMAGE.

Syntax

HDF_DF24_READREF, *Filename*, *Reference_number*

Arguments

Filename

A scalar string containing the name of the file to be read.

Reference_number

The reference number for a 24-bit raster image.

Keywords

None

Version History

4.0	Introduced

HDF_DF24_RESTART

The HDF_DF24_RESTART procedure causes the next call to HDF_DF24_GETIMAGE to read the first 24-bit image in the HDF file.

Syntax

HDF_DF24_RESTART

Arguments

None

Keywords

None

Version History

4.0	Introduced

HDF_DFAN_ADDFDS

The HDF_DFAN_ADDFDS procedure adds a file description to an HDF file.

Syntax

HDF_DFAN_ADDFDS, *Filename*, *Description*

Arguments

Filename

A scalar string containing the name of the file to be written.

Description

A string or a array of bytes containing the information to be written.

Keywords

None

Version History

4.0	Introduced

HDF_DFAN_ADDFID

The HDF_DFAN_ADDFID procedure adds a file annotation to an HDF file. A file can have multiple annotations added.

Syntax

HDF_DFAN_ADDFID, *Filename*, *Label*

Arguments

Filename

A scalar string containing the name of the file to be written.

Label

A string containing the annotation string.

Keywords

None

Examples

```
; Open the HDF file:
filename = 'FID.hdf'
hid = HDF_OPEN(filename,/CREATE)
; Write two file annotations:
HDF_DFAN_ADDFID, filename, 'File Annotation #1'
HDF_DFAN_ADDFID, filename, 'File Annotation #2'
; Read the two annotations back:
HDF_DFAN_GETFID, filename, fid1
HDF_DFAN_GETFID, filename, fid2
HELP, fid1, fid2
; Try to read a non-existent FID:
HDF_DFAN_GETFID, filename, fid3
; Read the FIRST fid again, using the FIRST keyword:
HDF_DFAN_GETFID, filename, fid4, /FIRST
HELP, fid4
; Close the HDF file:
HDF_CLOSE, hid
```

IDL Output

```
FID1            STRING    = 'File Annotation #1'
FID2            STRING    = 'File Annotation #2'

% HDF_DFAN_GETFID: Could not read ID length

FID4            STRING    = 'File Annotation #1'
```

Version History

4.0	Introduced

HDF_DFAN_GETDESC

The HDF_DFAN_GETDESC procedure reads the description for the given tag and reference number in an HDF file.

Syntax

HDF_DFAN_GETDESC, *Filename*, *Tag*, *Ref*, *Description* [, /STRING]

Arguments

Filename

A scalar string containing the name of the file to be read.

Tag

The tag number.

Reference

The reference number.

Description

A named variable in which the description is returned as a vector of bytes.

If a description does not exist, the *Description* variable will contain either a 0L (long-integer zero) or a blank string, and a warning message will be printed. Warning messages can be suppressed by setting the !QUIET system variable to a non-zero value.

Keywords

STRING

Set this keyword to return the description as a string rather than a vector of bytes.

Examples

```
desc1 = 'FILE DESCRIPTION NUMBER 1'
tag_image = 302
file = 'DEMOdesc.hdf'
fid = HDF_OPEN(file, /CREATE)
```

```
HDF_DFR8_ADDIMAGE, file, DIST(10)
HDF_DFAN_PUTDESC, file, tag_image, HDF_DFR8_LASTREF(), desc1
; Read the description and return a vector of bytes:
HDF_DFAN_GETDESC, file, tag_image, HDF_DFR8_LASTREF(), out_desc1
HELP, out_desc1
PRINT, STRING(out_desc1)
; Read the description and return an IDL string variable:
HDF_DFAN_GETDESC, file, tag_image, HDF_DFR8_LASTREF(), $
    out_desc2, /STRING
HELP, out_desc2
HDF_CLOSE, fid
```

IDL Output

```
OUT_DESC1       BYTE      = Array(25)

FILE DESCRIPTION NUMBER 1

OUT_DESC2       STRING    = 'FILE DESCRIPTION NUMBER 1'
```

Version History

4.0	Introduced

See Also

HDF_DFAN_PUTDESC

HDF_DFAN_GETFDS

The HDF_DFAN_GETFDS procedure reads the next available file description from an HDF file.

Syntax

HDF_DFAN_GETFDS, *Filename*, *Description* [, /FIRST] [, /STRING]

Arguments

Filename

A string containing the name of the file to be read.

Description

A named variable in which the description is returned. By default, the description is returned as a vector of bytes. Set the STRING keyword to return the description as a string.

If a description does not exist, the *Description* variable will contain either a 0L (long-integer zero) or a blank string, and a warning message will be printed. Warning messages can be suppressed by setting the !QUIET system variable to a non-zero value.

Keywords

FIRST

Set this keyword to read the first file description in the file. If FIRST is not set, the next available file description (which can be the first file description) will be read.

STRING

Set this keyword to return *Description* as a string instead of a vector of bytes.

Examples

```
filename = 'DEMOfds.hdf'
fds1 = 'FILE DESCRIPTION NUMBER 1'
fds2 = 'SHORT FDS 2'
; Create an HDF file:
```

```
fid = HDF_OPEN(filename, /CREATE)
; Add first file description:
HDF_DFAN_ADDFDS, filename, fds1
; Add second file description:
HDF_DFAN_ADDFDS, filename, fds2
; Get the first file description:
HDF_DFAN_GETFDS, filename, out_fds1, /FIRST
HELP, out_fds1
PRINT, STRING(out_fds1)
; Get the second file description:
HDF_DFAN_GETFDS, filename, out_fds2, /STRING
HELP, out_fds2
; Close the HDF file:
HDF_CLOSE, fid
```

IDL Output

```
OUT_FDS1        BYTE      = Array(25)

FILE DESCRIPTION NUMBER 1

OUT_FDS2        STRING    = 'SHORT FDS 2'
```

Version History

4.0	Introduced

See Also

HDF_DFAN_ADDFDS, HDF_DFAN_ADDFID, HDF_DFAN_GETDESC, HDF_DFAN_GETFID

HDF_DFAN_GETFID

The HDF_DFAN_GETFID procedure reads the next available file annotation from an HDF file.

Syntax

HDF_DFAN_GETFID, *Filename*, *Label* [, /FIRST]

Arguments

Filename

A scalar string containing the name of the file to be read.

Label

A named variable in which the annotation is returned as a string.

Keywords

FIRST

Set this keyword to read the first annotation in the file. Otherwise, the next available annotation is read (which may be the first annotation).

Examples

For an example using this routine, see "HDF_DFAN_ADDFID" on page 370.

Version History

4.0	Introduced

HDF_DFAN_GETLABEL

The HDF_DFAN_GETLABEL procedure reads the label for the given tag-reference pair in an HDF file.

Syntax

HDF_DFAN_GETLABEL, *Filename*, *Tag*, *Ref*, *Label*

Arguments

Filename

A scalar string that contains the name of the file to be read.

Tag

The tag number.

Reference

The reference number.

Label

A named variable in which the label is returned as a string.

Keywords

None

Examples

```
fid = HDF_OPEN('test.hdf', /ALL)
label = 'TEST LABEL'
tag = 105 ; The annotation tag.
ref = 2 ; Choose a reference number.
; Write the label:
HDF_DFAN_PUTLABEL, 'test.hdf', tag, ref, label
; Read back the label:
HDF_DFAN_GETLABEL, 'test.hdf', tag, ref, outl
HELP, outl ; They look the same...
; Close the HDF file:
HDF_CLOSE, fid
```

Version History

4.0	Introduced

See Also

HDF_DFAN_GETDESC, HDF_DFAN_LABLIST, HDF_DFAN_PUTDESC, HDF_DFAN_PUTLABEL

HDF_DFAN_LABLIST

The HDF_DFAN_LABLIST function retrieves a list of the reference numbers and the corresponding labels for a given tag in an HDF file.

Syntax

Result = HDF_DFAN_LABLIST(*Filename*, *Tag*, *Reflist*, *Labellist*
[, LISTSIZE=*value*] [, MAXLABEL=*value*] [, STARTPOS=*value*] [, /STRING])

Return Value

If successful, the number of entries found is returned.

Arguments

Filename

A scalar string containing the name of the file to be read.

Tag

The tag number.

Reflist

A named variable in which an array of reference numbers associated with the given tag is returned.

Labellist

A named variable in which an array of labels is returned. Unless the STRING keyword is set, *Labellist* will contain an N_ELEMENTS(*Reflist*) by MAXLABEL array of bytes. Note that array elements containing labels that are shorter than MAXLABEL will be padded with zeroes.

Keywords

LISTSIZE

Set the maximum size of the *Reflist* and *Labellist* returned. The default is to read all references present, or 20 if the inquiry to obtain the number of references fails.

MAXLABEL

Use this keyword to override the default label length of 16.

STARTPOS

Use this keyword to set the default starting position in the *Reflist* array.

STRING

Set this keyword to return an array of strings rather than an array of bytes. If STRING is set, the MAXLABEL keyword is ignored and full-length strings are returned.

Examples

```
tag_image = 302
file = 'DEMOlablist.hdf'
n_images = HDF_DFAN_LABLIST(file, tag_image, refs, list, /STRING)
help, n_images, refs, list
PRINT, list(0)
; Find all the compressed images:
tag_image_comp = 303
n_comp_images = HDF_DFAN_LABLIST(file, tag_image_comp, $
   refs, list, MAXLABEL=5)
HELP, n_comp_images, refs, list
```

IDL Output

```
N_IMAGES        LONG      =            2
REFS            INT       = Array(2)
LIST            STRING    = Array(2)

SAMPLE IMAGE LABEL

N_COMP_IMAGES   LONG      =            3
REFS            INT       = Array(3)
LIST            BYTE      = Array(5, 3)
```

Version History

4.0	Introduced

See Also

HDF_DFAN_GETLABEL, HDF_DFAN_PUTLABEL

HDF_DFAN_LASTREF

The HDF_DFAN_LASTREF function returns the reference number of the most recently read or written annotation in an HDF file.

Syntax

Result = HDF_DFAN_LASTREF( )

Return Value

Returns the reference number of the most recently read or written annotation.

Arguments

None

Keywords

None

Version History

4.0	Introduced

HDF_DFAN_PUTDESC

The HDF_DFAN_PUTDESC procedure writes a description for the given tag and reference number in an HDF file.

Syntax

HDF_DFAN_PUTDESC, *Filename*, *Tag*, *Ref*, *Description*

Arguments

Filename

A scalar string containing the name of the file to be written.

Tag

The tag number.

Reference

The reference number.

Description

A string or array of bytes containing the information to be written.

If a description does not exist, the *Description* variable will contain either a 0L (long-integer zero) or a blank string, and a warning message will be printed. Warning messages can be suppressed by setting the !QUIET system variable to a non-zero value.

Keywords

None

Examples

See the example for "HDF_DFAN_GETDESC" on page 372.

Version History

4.0	Introduced

HDF_DFAN_PUTLABEL

The HDF_DFAN_PUTLABEL procedure writes a label for the given tag and reference number in an HDF file.

Syntax

HDF_DFAN_PUTLABEL, *Filename*, *Tag*, *Ref*, *Label*

Arguments

Filename

A scalar string containing the name of the file to be written.

Tag

The tag number.

Ref

The reference number.

Label

A string containing the description to write.

Keywords

None

Version History

4.0	Introduced

HDF_DFP_ADDPAL

The HDF_DFP_ADDPAL procedure appends a palette to an HDF file.

Syntax

HDF_DFP_ADDPAL, *Filename*, *Palette*

Arguments

Filename

A scalar string containing the name of the file to be written.

Palette

A vector or array containing palette data. Palettes must be either [3, 256] arrays or 786-element vectors.

Keywords

None

Version History

4.0	Introduced

HDF_DFP_GETPAL

The HDF_DFP_GETPAL procedure reads the next available palette from an HDF file.

Syntax

HDF_DFP_GETPAL, *Filename*, *Palette*

Arguments

Filename

A scalar string containing the name of the file to be read.

Palette

A named variable in which the palette data is returned.

Keywords

None

Version History

4.0	Introduced

HDF_DFP_LASTREF

The HDF_DFP_LASTREF function returns the reference number of the most recently read or written palette in an HDF file.

Syntax

Result = HDF_DFP_LASTREF( )

Return Value

Returns the reference number of the most recently read or written palette.

Arguments

None

Keywords

None

Version History

4.0	Introduced

HDF_DFP_NPALS

The HDF_DFP_NPALS function returns the number of palettes present in an HDF file. This number includes palettes associated with RIS8 (8-bit raster) images.

Syntax

Result = HDF_DFP_NPALS(*Filename*)

Return Value

Returns the number of palettes.

Arguments

Filename

A scalar string containing the name of the desired HDF file.

Keywords

None

Version History

4.0	Introduced

HDF_DFP_PUTPAL

The HDF_DFP_PUTPAL procedure appends a palette to an HDF file.

Syntax

HDF_DFP_PUTPAL, *Filename*, *Palette* [, /DELETE] [, /OVERWRITE]

Arguments

Filename

A scalar string containing the name of the file to be written.

Palette

A vector or array containing palette data. Palettes must be either [3, 256] arrays or 786-element vectors.

Keywords

DELETE

Set this keyword to delete the HDF file (if it exists) and create a new HDF file with the specified palette as its first object.

Note

The HDF file must be closed before the DELETE keyword is specified. Attempting to delete an open HDF file will result in an error.

OVERWRITE

Set this keyword to overwrite the previous palette with the one specified by *Palette*.

Examples

```
; Create HDF file:
id = HDF_OPEN('test.hdf', /CREATE, /RDWR)
; Add a palette:
HDF_DFP_PUTPAL,'test.hdf'’, FINDGEN(3,256)
; Print number of palettes:
PRINT, HDF_DFP_NPALS('test.hdf')
```

```
; Append a palette:
HDF_DFP_PUTPAL,’test.hdf’,findgen(3,256)
; Print the number of palettes:
PRINT, HDF_DFP_NPALS('test.hdf')
; Overwrite the last palette:
HDF_DFP_PUTPAL, 'test.hdf', FINDGEN(3,256), /OVERWRITE
; Print the number of palettes:
PRINT, HDF_DFP_NPALS('test.hdf')
; An attempt to delete a file and add a new palette
; without first closing the HDF file fails:
HDF_DFP_PUTPAL, 'test.hdf', $
   FINDGEN(3,256), /DELETE
; Close the HDF file:
HDF_CLOSE, id
; Delete file and add a new palette:
HDF_DFP_PUTPAL, 'test.hdf', FINDGEN(3,256), /DELETE
; Print the number of palettes:
PRINT, HDF_DFP_NPALS('test.hdf')
```

IDL Output

```
1

2

2

% HDF_DFP_PUTPAL: Could not write palette
% Execution halted at:  $MAIN$

1
```

Version History

4.0	Introduced

HDF_DFP_READREF

The HDF_DFP_READREF procedure sets the reference number of the palette in an HDF file to be read by the next call to HDF_DFP_GETPAL.

Syntax

HDF_DFP_READREF, *Filename*, *Reference_number*

Arguments

Filename

A scalar string containing the name of the file to be read.

Reference_number

The reference number of a palette.

Keywords

None

Version History

4.0	Introduced

HDF_DFP_RESTART

The HDF_DFP_RESTART procedure causes the next call to HDF_DFR8_GETPAL to read from the first palette in an HDF file.

Syntax

HDF_DFP_RESTART

Arguments

None

Keywords

None

Version History

4.0	Introduced

HDF_DFP_WRITEREF

The HDF_DFP_WRITEREF procedure sets the reference number for the next palette to be written to an HDF file. Normally, the HDF library automatically chooses a reference number for the palette. This procedure allows you to override that choice.

Syntax

HDF_DFP_WRITEREF, *Filename*, *Reference_number*

Arguments

Filename

A scalar string containing the name of the file to be read.

Reference_number

The new reference number.

Keywords

None

Version History

4.0	Introduced

HDF_DFR8_ADDIMAGE

The HDF_DFR8_ADDIMAGE procedure appends an 8-bit raster image to the specified HDF file.

Note

Input data is converted to bytes before being written to the file, as images in the DFR8 HDF model are necessarily byte images.

Syntax

HDF_DFR8_ADDIMAGE, *Filename*, *Image* [, /FORCE_BASELINE{useful only if QUALITY<25}] [, /JPEG | , /RLE] [[, /IMCOMP] , PALETTE=*vector or array*] [, QUALITY=*value*]

Arguments

Filename

A scalar string containing the name of the file to be written.

Image

A two-dimensional array containing the image data. If this array is not byte-type data, it is converted to bytes before writing.

Keywords

FORCE_BASELINE

Set this keyword to force the JPEG quantization tables to be constrained to the range 1...255. This provides full baseline compatibility with external JPEG applications, but only makes a difference if the QUALITY keyword is set to a value less than 25. The default is TRUE.

JPEG

Set this keyword to compress the image being added using the JPEG (Joint Photographic Expert Group) method. Note that JPEG compression is *lossy*; see WRITE_JPEG in the *IDL Reference Guide* for more information about when this method is appropriate. (In other words, using JPEG compression to reduce the size of an images changes the values of the pixels and hence may alter the meaning of the corresponding data.) Setting either the QUALITY or the FORCE_BASELINE keywords implies this method.

IMCOMP

Set this keyword to store the image using imcomp data compression. Note that you *must* specify a palette. Note also that the JPEG and RLE compression methods are far superior; imcomp data compression should only be used if the images will be viewed on monitors with a very small number of colors (monochrome or 16-color).

PALETTE

Set this keyword to a vector or array containing valid palette data. Palettes must be either [3, 256] arrays or 786-element vectors. Set PALETTE equal to zero to specify that no palette be used. If the PALETTE keyword is not specified, the current palette (which may be no palette, if a palette has not been specified elsewhere or if the null palette has been explicitly specified with HDF_DFR8_SETPALETTE) will be used.

Note that if a palette is specified, it becomes the current palette, even if a default palette has been specified with HDF_DFR8_SETPALETTE.

Note also that if IMCOMP data reduction is used, you *must* specify a valid palette with the PALETTE keyword. It is not sufficient to set the current palette via other means.

QUALITY

Set this keyword equal to the JPEG “quality” desired. This value should be in range 0 (terrible image quality but excellent compression) to 100 (excellent image quality but minimum compression). The default is 75. Setting this keyword implies that the JPEG keyword is set. Lower values of QUALITY produce higher compression ratios and smaller files.

RLE

Set this keyword to store the image using run length compression. RLE compression is lossless, and is recommended for images where data retention is critical.

Examples

Assuming that we start with a file, `new.hdf`, with no 8-bit raster images, images could be appended and overwritten, with the following commands:

```
; Write the first image to the file:
```

```
HDF_DFR8_ADDIMAGE, 'new.hdf', Image1
; Append 2nd image:
HDF_DFR8_ADDIMAGE, 'new.hdf', Image2
; Append 3rd image:
HDF_DFR8_ADDIMAGE, 'new.hdf', Image3
; Use HDF_DFR8_PUTIMAGE to erase all previous images and
; write a new image at the first position in the file:
HDF_DFR8_PUTIMAGE, 'new.hdf', Image4
; Append 2nd image:
HDF_DFR8_ADDIMAGE, 'new.hdf', Image5
```

Version History

4.0	Introduced

See Also

HDF_DFR8_GETIMAGE, HDF_DFR8_PUTIMAGE, WRITE_JPEG

HDF_DFR8_GETIMAGE

The HDF_DFR8_GETIMAGE procedure retrieves an image and optionally, its palette, from an HDF file.

Syntax

HDF_DFR8_GETIMAGE, *Filename*, *Image* [, *Palette*]

Arguments

Filename

A scalar string containing the name of the file to be read.

Image

A named variable in which the image is returned.

Palette

A named variable in which the palette is returned as a 3-element by 256-element byte array. If the image does not have an associated palette, this variable is returned as 0.

Keywords

None

Version History

4.0	Introduced

HDF_DFR8_GETINFO

The HDF_DFR8_GETINFO procedure retrieves information about the current 8-bit HDF image.

Syntax

HDF_DFR8_GETINFO, *Filename*, *Width*, *Height*, *Has_Palette*

Arguments

Filename

A string containing the name of the file to be read.

Width

A named variable in which the width of the image is returned.

Height

A named variable in which the height of the image is returned.

Has_Palette

A named variable in which 1 is returned if a palette is present. Otherwise, 0 is returned.

Keywords

None

Examples

```
; Open the file myhdf.hdf:
h = HDF_OPEN('myhdf.hdf')
; Retrieve info about an image:
HDF_DFR8_GETINFO, 'myhdf.hdf', width, height, has_palette
; Print info about returned variables:
HELP, width, height, has_palette
; Close the HDF file:
HDF_CLOSE('myhdf.hdf')
```

IDL Output

```
WIDTH LONG = 536
HEIGHT LONG = 412
HAS_PALETTE LONG = 1
```

Example Code

For a more detailed example, see the file `hdf_info.pro`, located in the `examples/doc/sdf` subdirectory of the IDL distribution.

Version History

4.0	Introduced

See Also

HDF_DFR8_GETIMAGE, HDF_DFR8_NIMAGES, HDF_DFR8_READREF, HDF_DFR8_RESTART

HDF_DFR8_LASTREF

The HDF_DFR8_LASTREF function returns the reference number of the most recently read or written 8-bit image in an HDF file.

Syntax

Result = HDF_DFR8_LASTREF( )

Return Value

Returns the reference number of the most recently read or written image.

Arguments

None

Keywords

None

Examples

```
h = HDF_OPEN('myhdf.hdf') ; Open an hdf file.
; IDL prints "0", meaning that the call was successful,
; but no reference number was available:
PRINT, HDF_DFR8_LASTREF()
; Create a 2D array representing an 8-bit image:
a = BINDGEN(100,100)
; Write the image to the file:
HDF_DFR8_ADDIMAGE, 'myhdf.hdf', a
; IDL prints the reference number for the last 8-bit image
; operation (for example, "2"). Note the reference number
; is not simply a 1-based "image number"; it could easily be
; "2" or "3" for the first operation on the file:
PRINT, HDF_DFR8_LASTREF()
HDF_DFR8_ADDIMAGE, 'myhdf.hdf', a ; Add another image.
; IDL prints "2", because we’ve put two 8-bit images in the file:
PRINT, HDF_DFR8_NIMAGES('myhdf.hdf')
HDF_CLOSE, h ; Close the file.
```

Version History

4.0	Introduced

See Also

HDF_DFR8_ADDIMAGE, HDF_DFR8_GETIMAGE, HDF_DFR8_GETINFO, HDF_DFR8_LASTREF, HDF_DFR8_NIMAGES, HDF_DFR8_READREF, HDF_DFR8_RESTART

HDF_DFR8_NIMAGES

The HDF_DFR8_NIMAGES function returns the number of 8-bit images in the specified HDF file.

Syntax

Result = HDF_DFR8_NIMAGES(*Filename*)

Return Value

Returns the number of 8-bit images in the given HDF file. The function returns -1 if the specified file is invalid or damaged.

Arguments

Filename

A string containing the name of the file to be read.

Keywords

None

Examples

```
; Open the file myhdf.hdf:
h = HDF_OPEN('myhdf.hdf')
; Retrieve the number of 8-bit images in the file into a variable:
number = HDF_DFR8_NIMAGES('myhdf.hdf')
HDF_CLOSE, h ; Close the file.
```

Version History

4.0	Introduced

See Also

HDF_DFR8_GETIMAGE, HDF_DFR8_GETINFO, HDF_DFR8_READREF
HDF_DFR8_RESTART

HDF_DFR8_PUTIMAGE

The HDF_DFR8_PUTIMAGE procedure writes an 8-bit raster image as the first image in an HDF file. If there are images in the file, this procedure erases all other 8-bit and 24-bit images and writes *Image* as the first image in the file.

Note

Input data is converted to bytes before being written to the file, as images in the DFR8 HDF model are necessarily byte images.

Syntax

HDF_DFR8_PUTIMAGE, *Filename*, *Image* [, /FORCE_BASELINE{useful only if QUALITY<25}] [[, /IMCOMP] , PALETTE=*vector or array*] [, /JPEG | , /RLE] [, QUALITY=*value*]

Arguments

Filename

A scalar string containing the name of the file to be written.

Image

A two-dimensional array containing the image data. If this array is not byte-type data, it is converted to bytes before writing.

Keywords

FORCE_BASELINE

Set this keyword to force the JPEG quantization tables to be constrained to the range 1...255. This provides full baseline compatibility with external JPEG applications, but only makes a difference if the QUALITY keyword is set to a value less than 25. The default is TRUE.

JPEG

Set this keyword to compress the image being added using the JPEG (Joint Photographic Expert Group) method. Note that JPEG compression is *lossy*; see WRITE_JPEG in the *IDL Reference Guide* for more information about when this method is appropriate. (In other words, using JPEG compression to reduce the size of an images changes the values of the pixels and hence may alter the meaning of the corresponding data.) Setting either the QUALITY or the FORCE_BASELINE keywords implies this method.

IMCOMP

Set this keyword to store the image using imcomp data compression. Note that you *must* specify a palette. Note also that the JPEG and RLE compression methods are far superior; imcomp data compression should only be used if the images will be viewed on monitors with a very small number of colors (monochrome or 16-color).

PALETTE

Set this keyword to a vector or array containing valid palette data. Palettes must be either [3, 256] arrays or 786-element vectors. Set PALETTE equal to zero to specify that no palette be used. If the PALETTE keyword is not specified, the current palette (which may be no palette, if a palette has not been specified elsewhere or if the null palette has been explicitly specified with HDF_DFR8_SETPALETTE) will be used.

Note that if a palette is specified, it becomes the current palette, even if a default palette has been specified with HDF_DFR8_SETPALETTE.

Note also that if IMCOMP data reduction is used, you *must* specify a valid palette with the PALETTE keyword. It is not sufficient to set the current palette via other means.

QUALITY

Set this keyword equal to the JPEG “quality” desired. This value should be in range 0 (terrible image quality but excellent compression) to 100 (excellent image quality but minimum compression). The default is 75. Setting this keyword implies that the JPEG keyword is set. Lower values of QUALITY produce higher compression ratios and smaller files.

RLE

Set this keyword to store the image using run length compression. RLE compression is lossless, and is recommended for images where data retention is critical.

Version History

4.0	Introduced

See Also

HDF_DFR8_ADDIMAGE, HDF_DFR8_GETIMAGE, WRITE_JPEG

HDF_DFR8_READREF

The HDF_DFR8_READREF procedure sets the reference number of the image to be read from an HDF file by the next call to HDF_DFR8_GETIMAGE.

Syntax

HDF_DFR8_READREF, *Filename*, *Reference_number*

Arguments

Filename

A scalar string containing the name of the file to be read.

Reference_number

A reference number for an 8-bit raster image.

Keywords

None

Version History

4.0	Introduced

HDF_DFR8_RESTART

The HDF_DFR8_RESTART procedure causes the next call to HDF_DFR8_GETIMAGE to read from the first image in the HDF file.

Syntax

HDF_DFR8_RESTART

Arguments

None

Keywords

None

Version History

4.0	Introduced

HDF_DFR8_SETPALETTE

The HDF_DFR8_SETPALETTE procedure sets the current palette to be used for subsequent images in an HDF file. The current palette will be used when adding images with the HDF_DFR8_ADDIMAGE routine.

Syntax

HDF_DFR8_SETPALETTE, *Palette*

Arguments

Palette

A 768-element byte array of palette data. This array be a vector (e.g., `BYTARR(768)`) or a two-dimensional array (e.g., `BYTARR(3, 256)`).

Set the Palette array to the integer zero to set the current palette to no palette.

Keywords

None

Version History

4.0	Introduced

HDF_DUPDD

The HDF_DUPDD procedure generates new references to existing data in an HDF file.

Syntax

HDF_DUPDD, *FileHandle*, *NewTag*, *NewRef*, *OldTag*, *OldRef*

Arguments

FileHandle

The HDF file handle returned from a previous call to HDF_OPEN.

NewTag

An integer tag for new data descriptor.

NewRef

An integer reference number for the new data descriptor.

OldTag

The integer tag of data descriptor to duplicate.

OldRef

The reference number of data descriptor to duplicate.

Keywords

None

Version History

4.0	Introduced

HDF_EXISTS

The HDF_EXISTS function returns True if the HDF scientific data format library is supported on the current IDL platform.

This routine is written in the IDL language. Its source code can be found in the file `hdf_exists.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = HDF_EXISTS( )

Return Value

Returns a 1 (True) if the library is supported or a 0 (False) if the library is not supported.

Arguments

None

Keywords

None

Examples

The following IDL command prints an error message if the HDF library is not available:

```
IF HDF_EXISTS() EQ 0 THEN PRINT, 'HDF not supported.'
```

Version History

4.0	Introduced

HDF_GR_ATTRINFO

This function retrieves the name, data type, and number of values of the attribute for the HDF data object identified by the parameter obj_id.

Syntax

Result = HDF_GR_ATTRINFO(*obj_id*, *attr_index*, *name*, *data_type*, *count*)

Return Value

Returns SUCCEED (0) if successful or FAIL (-1) otherwise.

Arguments

obj_id

Raster image identifier (ri_id), returned by HDF_GR_CREATE or HDF_GR_SELECT, or HDF GR interface identifier (gr_id), returned by HDF_GR_START.

attr_index

Index of the attribute. The value of this parameter can be obtained using HDF_GR_FINDATTR, HDF_GR_NAMETOINDEX or HDF_GR_REFTOINDEX, depending on available information. Valid values range from 0 to the total number of attributes attached to the object minus 1. The total number of attributes attached to the file can be obtained using the routine HDF_GR_FILEINFO. The total number of attributes attached to an image can be obtained using the routine HDF_GR_GETIMINFO.

name

A named variable in which the name of the attribute is returned.

data_type

A named variable in which the attribute data type is returned. See “IDL and HDF Data Types” on page 317.

count

A named variable in which the number of attributes is returned.

Keywords

None

Version History

4.0	Introduced

HDF_GR_CREATE

This function creates an HDF GR raster image. Once a raster image has been created, it is not possible to change its name, data type, dimension sizes or number of pixel components. However, it is possible to create a raster image and close the file before writing any data values to it. Later, the values can be added to or modified in the raster image, which then can be obtained using HDF_GR_SELECT.

Note

On creation, any interlace mode may be set. This mode will be used until the file is closed. If the resulting file is reopened, the interlace mode will revert to pixel-interlace (0). Data can still be read in any interlace mode using the INTERLACE keyword to HDF_GR_READIMAGE. This is a limitation of the current HDF library.

Syntax

Result = HDF_GR_CREATE(*gr_id*, *name*, *ncomp*, *data_type*, *interlace_mode*, *dim_sizes*)

Return Value

Returns a raster image identifier if successful or FAIL (-1) otherwise.

Arguments

gr_id

GR interface identifier returned by HDF_GR_START.

name

Name of the raster image. The length of the name should not be longer than 256 characters.

ncomp

Number of pixel components in the image. This parameter must have a value of at least 1.

data_type

Type of the image data. This parameter can be any of the data types supported by the HDF library. See "IDL and HDF Data Types" on page 317.

interlace_mode

Interlace mode of the image data. Valid values are:

- 0 = Pixel interlace
- 1 = Line interlace
- 2 = Component interlace

dim_sizes

Array of sizes for each dimension of the image. The dimensions must be specified and their values must be greater than 0.

Keywords

None

Version History

5.2	Introduced

HDF_GR_END

This procedure terminates the HDF GR interface session identified by the parameter *gr_id*. HDF_GR_END, together with HDF_GR_START, define the extent of a HDF GR interface session. HDF_GR_END disposes of the internal structures initialized by the corresponding call to HDF_GR_START. There must be a call to HDF_GR_END for each call to HDF_GR_START; failing to provide one may cause loss of data. HDF_GR_START and HDF_GR_END do not manage file access; use HDF_OPEN and HDF_CLOSE to open and close HDF files. HDF_OPEN must be called before HDF_GR_START and HDF_CLOSE must be called after HDF_GR_END. Failure to properly close the HDF file with HDF_GR_END and HDF_CLOSE may result in lost data or corrupted HDF files.

Syntax

HDF_GR_END, *gr_id*

Arguments

gr_id

HDF GR interface identifier returned by HDF_GR_START.

Keywords

None

Version History

5.2	Introduced

HDF_GR_ENDACCESS

This procedure terminates access to the raster image identified by the parameter *ri_id* and disposes of the raster image identifier. This access is initiated by either HDF_GR_SELECT or HDF_GR_CREATE. There must be a call to HDF_GR_ENDACCESS for each call to HDF_GR_SELECT or HDF_GR_CREATE; failing to provide this will result in loss of data.

Syntax

HDF_GR_ENDACCESS, *ri_id*

Arguments

ri_id

Raster image identifier returned by HDF_GR_CREATE or HDF_GR_SELECT.

Keywords

None

Version History

5.2	Introduced

HDF_GR_FILEINFO

This function retrieves the number of raster images and the number of global attributes for the HDF GR interface identified by the parameter *gr_id*, and stores them into the parameters *n_images* and *n_file_attrs*, respectively. The term "global attributes" refers to attributes that are assigned to the file instead of individual raster images. These attributes are created by HDF_GR_SETATTR with the object identifier parameter set to a HDF GR interface identifier (*gr_id*) rather than a raster image identifier (*ri_id*). HDF_GR_FILEINFO is useful in finding the range of acceptable indices for HDF_GR_SELECT calls.

Syntax

Result = HDF_GR_FILEINFO(*gr_id*, *n_images*, *n_file_attrs*)

Return Value

Returns SUCCEED (or 0) if successful or FAIL (-1) otherwise.

Arguments

gr_id

HDF GR interface identifier returned by HDF_GR_START.

n_images

A named variable that will contain the number of raster images in the file.

n_file_attrs

A named variable that will contain the number of global attributes in the file.

Keywords

None

Version History

5.2	Introduced

HDF_GR_FINDATTR

This function finds the index of an HDF data object's attribute given its attribute name. HDF_GR_FINDATTR returns the index of the attribute whose name is specified by the parameter *attr_name* for the object identified by the parameter *obj_id*.

Syntax

Result = HDF_GR_FINDATTR(*obj_id*, *attr_name*)

Return Value

Returns the index of the attribute if successful or FAIL (-1) otherwise.

Arguments

obj_id

Raster image identifier (*ri_id*), returned by HDF_GR_CREATE or HDF_GR_SELECT, or HDF GR interface identifier (*gr_id*), returned by HDF_GR_START.

attr_name

Name of the attribute.

Keywords

None

Version History

5.2	Introduced

HDF_GR_GETATTR

This function obtains all values of the HDF GR attribute that is specified by its index, *attr_index*, and is attached to the object identified by the parameter *obj_id*.

Syntax

Result = HDF_GR_GETATTR(*obj_id*, *attr_index*, *values*)

Return Value

Returns SUCCEED (0) if successful or FAIL (-1) otherwise.

Arguments

obj_id

Raster image identifier (*ri_id*), returned by HDF_GR_CREATE or HDF_GR_SELECT, or HDF GR interface identifier (*gr_id*), returned by HDF_GR_START.

attr_index

Index of the attribute.

The value of the parameter *attr_index* can be obtained by using HDF_GR_FINDATTR, HDF_GR_NAMETOINDEX, or HDF_GR_REFTOINDEX, depending on available information. Valid values of *attr_index* range from 0 to the total number of attributes of the object - 1. The total number of attributes attached to the file can be obtained using the routine HDF_GR_FILEINFO. The total number of attributes attached to the image can be obtained using the routine HDF_GR_GETIMINFO. HDF_GR_GETATTR only reads all values assigned to the attribute and not a subset.

values

A named variable that will contain the attribute values.

Keywords

None

Version History

5.2	Introduced

HDF_GR_GETCHUNKINFO

This function retrieves chunking information about the HDF GR raster image identified by the parameter *ri_id* into the parameters *dim_length* and *flag*. Note that only chunk dimensions are retrieved; compression information is not available with this function.

Syntax

Result = HDF_GR_GETCHUNKINFO(*ri_id*, *dim_length*, *flag*)

Return Value

Returns SUCCEED (0) if successful or FAIL (-1) otherwise.

Arguments

ri_id

Raster image identifier returned by HDF_GR_CREATE or HDF_GR_SELECT.

dim_length

A named variable that will contain the array of chunk dimensions.

flag

A named variable that will contain the compression/chunk flag.

The value returned in the parameter *flag* indicates if the raster image is not chunked, chunked, or chunked and compressed. The following table shows the possible values of the parameter *flag* and the corresponding characteristics of the raster image.

Values of flag = Raster Image Characteristics

- -1 = Not chunked
- 0 = Chunked and not compressed
- 1 = Chunked and compressed with either the run-length encoding (RLE), Skipping Huffman or GZIP compression algorithms

Keywords

None

Version History

5.2	Introduced

HDF_GR_GETIMINFO

This function retrieves general information about an HDF GR raster image. HDF_GR_GETIMINFO retrieves the name, number of components, data type, interlace mode, dimension sizes, and number of attributes of the raster image identified by the parameter *ri_id*. It also retrieves the number of attributes attached to the image into the parameter *num_attrs*.

Syntax

Result = HDF_GR_GETIMINFO(*ri_id*, *gr_name*, *ncomp*, *data_type*, *interlace_mode*, *dim_sizes*, *num_attrs*)

Return Value

Returns SUCCEED (0) if successful or FAIL (-1) otherwise.

Arguments

ri_id

Raster image identifier returned by HDF_GR_CREATE or HDF_GR_SELECT.

gr_name

A named variable that will contain the name of the raster image.

ncomp

A named variable that will contain the number of components in the raster image.

data_type

A named variable that will contain the data type of the raster image data. The valid values of the parameter *data_type* are listed in "IDL and HDF Data Types" on page 317.

interlace_mode

A named variable that will contain the interlace mode of the stored raster image data.

- 0 = Pixel interlace
- 1 = Line interlace

- 2 = Component interlace

dim_sizes

A named variable that will contain the sizes of the raster image dimensions.

num_attrs

A named variable that will contain the number of attributes attached to the raster image.

Keywords

None

Version History

5.2	Introduced

HDF_GR_GETLUTID

This function gets the identifier of the HDF GR palette attached to the raster image identified by the parameter *ri_id*.

Syntax

Result = HDF_GR_GETLUTID(*ri_id*, *pal_index*)

Return Value

Returns the palette identifier if successful or FAIL (-1) otherwise.

Arguments

ri_id

Raster image identifier returned by HDF_GR_CREATE or HDF_GR_SELECT.

pal_index

Index of the palette. Currently, only one palette can be assigned to a raster image, which means that pal_index should always be set to 0.

Keywords

None

Version History

5.2	Introduced

HDF_GR_GETLUTINFO

This function retrieves the number of pixel components, data type, interlace mode, and number of color lookup table entries of the palette identified by the parameter *pal_id*.

Syntax

Result = HDF_GR_GETLUTINFO(*pal_id*, *ncomp*, *data_type*, *interlace_mode*, *num_entries*)

Return Value

Returns SUCCEED (0) if successful or FAIL (-1) otherwise.

Arguments

pal_id

Palette identifier returned by HDF_GR_GETLUTID.

ncomp

A named variable in which the number of components in the palette is returned.

data_type

A named variable in which the HDF data type of the palette is returned. See "IDL and HDF Data Types" on page 317 for a description of the HDF data types.

interlace_mode

A named variable in which the interlace mode of the stored palette data is returned.

- 0 = Pixel interlace
- 1 = Line interlace
- 2 = Component interlace

num_entries

A named variable in which the number of color lookup table entries in the palette is returned.

Keywords

None

Version History

5.2	Introduced

HDF_GR_IDTOREF

This function returns the HDF reference number of the raster image identified by the parameter *ri_id*. This routine is commonly used for the purpose of annotating the raster image or including the raster image within an HDF Vgroup. The tag number for a GR is 306.

Syntax

Result = HDF_GR_IDTOREF(*ri_id*)

Return Value

Returns the HDF reference number of the raster image if successful or not found (0) otherwise.

Arguments

ri_id

Raster image identifier returned by HDF_GR_SELECT or HDF_GR_CREATE.

Keywords

None

Version History

5.2	Introduced

HDF_GR_LUTTOREF

This function returns the HDF reference number of the palette identified by the parameter *pal_id*. This function is commonly used for the purpose of annotating the palette or including the palette within a HDF Vgroup.

Syntax

Result = HDF_GR_LUTTOREF(*pal_id*)

Return Value

Returns the reference number of the palette if successful or not found (0) otherwise.

Arguments

pal_id

Palette identifier returned by HDF_GR_GETLUTID.

Keywords

None

Version History

5.2	Introduced

HDF_GR_NAMETOINDEX

This function returns the index of the raster image named *gr_name* for the HDF GR interface identified by the parameter *gr_id*.

The value of index can be passed into HDF_GR_SELECT to obtain the raster image identifier (ri_id).

Syntax

Result = HDF_GR_NAMETOINDEX(*gr_id*, *gr_name*)

Return Value

Returns the index of the raster image if successful or FAIL (-1) otherwise.

Arguments

gr_id

HDF_GR_ interface identifier returned by HDF_GR_START.

gr_name

Name of the raster image.

Keywords

None

Version History

5.2	Introduced

HDF_GR_READIMAGE

This function reads the subsample of the HDF GR raster image specified by the parameter *ri_id* into the variable *data*.

Syntax

Result = HDF_GR_READIMAGE(*ri_id*, *data* [, EDGE=*array*] [, /INTERLACE] [, START=*array*] [, STRIDE=*array*])

Return Value

Returns the specified variable containing the image subsample.

Arguments

ri_id

Raster image identifier returned by HDF_GR_CREATE or HDF_GR_SELECT

data

A named variable that will contain the image data.

Keywords

EDGE

Array specifying the number of values to be read along each dimension. The default is to read the entire specified image.

INTERLACE

Set this keyword to force data to be returned in INTERLACE mode. The default is pixel-interlacing (0) other possible values are 1 (line) and 2 (component).

START

Array specifying the starting location from where raster image data is read. Valid values of each element in the array are 0 to the size of the corresponding raster image dimension minus 1. The default is to read starting at the first pixel in each dimension (start = [0,0]).

STRIDE

Array specifying the interval between the values that will be read along each dimension. The default is for contiguous reading along each dimension (stride = [1,1]).

Note

The correspondence between the elements in the array *start* and the array *data* dimensions in the HDF GR interface is different from that in the HDF SD interface. The array *stride* specifies the reading pattern along each dimension. For example, if one of the elements of the array stride is 1, then every element along the corresponding dimension of the array data will be read. If one of the elements of the array *stride* is 2, then every other element along the corresponding dimension of the array data will be read, and so on. The correspondence between elements of the array *stride* and the dimensions of the array *data* is the same as described above for the array *start*. Each element of the array *edges* specifies the number of data elements to be read along the corresponding dimension. The correspondence between the elements of the array *edges* and the dimensions of the array *data* is the same as described above for the array *start*.

Version History

5.2	Introduced

HDF_GR_READLUT

This function reads the palette specified by the parameter *pal_id* into the *pal_data* variable.

Syntax

Result = HDF_GR_READLUT(*pal_id*, *pal_data* [, /INTERLACE])

Return Value

Returns SUCCEED (0) if successful or FAIL (-1) otherwise.

Arguments

pal_id

Palette identifier returned by HDF_GR_GETLUTID.

pal_data

A named variable that will contain the palette data.

Keywords

INTERLACE

Set this keyword to force pal_data to be returned in INTERLACE mode. The default is pixel-interlacing (0) other possible values are 1 (line) and 2 (component).

Version History

5.2	Introduced

HDF_GR_REFTOINDEX

This function returns the index of the HDF GR raster image specified by the parameter *gr_ref*.

Syntax

Result = HDF_GR_REFTOINDEX(*gr_id*, *gr_ref*)

Return Value

Returns the index of the image if successful or FAIL (-1) otherwise.

Arguments

gr_id

HDF GR interface identifier returned by HDF_GR_START.

gr_ref

Reference number of the raster image.

Keywords

None

Version History

5.2	Introduced

HDF_GR_SELECT

This function obtains the identifier of the HDF GR raster image specified by its index.

Syntax

Result = HDF_GR_SELECT(*gr_id*, *index*)

Return Value

Returns the raster image identifier if successful or FAIL (-1) otherwise.

Arguments

gr_id

HDF GR interface identifier returned by HDF_GR_START.

index

Index of the raster image in the file. Valid values range from 0 to the total number of raster images in the file minus 1. The total number of the raster images in the file can be obtained by using HDF_GR_FILEINFO.

Keywords

None

Version History

5.2	Introduced

HDF_GR_SETATTR

This function attaches the attribute to the HDF GR object specified by the parameter *obj_id*. The attribute is defined by its name, data type, number of attribute values, and the attribute values. HDF_GR_SETATTR provides a generic way for users to define metadata. It implements the label = value data abstraction. If an HDF GR interface identifier (*gr_id*) is specified as the parameter *obj_id*, a global attribute is created that applies to all objects in the file. If a raster image identifier (*ri_id*) is specified as the parameter *obj_id*, an attribute is attached to the specified raster image. Attribute values are passed in the parameter values. The number of attribute values is defined by the parameter count. If more than one value is stored, all values must have the same data type. If an attribute with the given name, data type and number of values exists, it will be overwritten. Currently, the only predefined attribute is the fill value, identified by the attribute name “FillValue”.

Syntax

Result = HDF_GR_SETATTR(*obj_id*, *attr_name*, *data_type*, *count*, *values*)

Return Value

Returns SUCCEED (0) if successful or FAIL (-1) otherwise.

Arguments

obj_id

Raster image identifier (ri_id), returned by HDF_GR_CREATE or HDF_GR_SELECT or HDF GR interface identifier (gr_id), returned by HDF_GR_START.

attr_name

Name of the attribute (string).

data_type

Data type of the attribute (integer). Can be any data type supported by the HDF library. These data types are listed under “IDL and HDF Data Types” on page 317.

count

Number of values in the attribute.

values

The attribute value.

Keywords

None

Version History

5.2	Introduced

HDF_GR_SETCHUNK

This function makes the HDF GR raster image specified by the parameter *ri_id* a chunked raster image according to the chunking and compression information provided in the parameters *comp_type* and *comp_prm*. Data can be compressed using run-length encoding (RLE), Skipping Huffman or GZIP compression algorithms.

Syntax

Result = HDF_GR_SETCHUNK(*ri_id*, *dim_length*, *comp_type*, *comp_prm*)

Return Value

Returns SUCCEED (or 0) if successful or FAIL (-1) otherwise.

Arguments

ri_id

Raster image identifier returned by HDF_GR_CREATE or HDF_GR_SELECT.

dim_length

Chunk dimensions array.

comp_type

Type of compression. Valid types are:

- 0 = uncompressed data
- 1 = data compressed using the RLE compression algorithm
- 3 = data compressed using the Skipping Huffman compression algorithm
- 4 = data compressed using the GZIP compression algorithm.

comp_prm

Compression parameters array. Specifies the compression parameters for the Skipping Huffman and GZIP compression methods. It contains only one element, which is set to the skipping size for Skipping Huffman compression or the deflate level for GZIP compression (1-9).

Keywords

None

Version History

5.2	Introduced

HDF_GR_SETCHUNKCACHE

This function allows the user to set the maximum number of chunks to be cached (*maxcache*). If HDF_GR_SETCHUNKCACHE is not called, *maxcache* is set to the number of chunks along the fastest changing dimension.

Syntax

Result = HDF_GR_SETCHUNKCACHE(*ri_id*, *maxcache*, *flags*)

Return Value

Returns the value of *maxcache* if successful or FAIL (-1) otherwise.

Arguments

ri_id

Raster image identifier returned by HDF_GR_CREATE or HDF_GR_SELECT.

maxcache

Maximum number of chunks to cache.

flags

Currently, the only HDF allowed value for flags is zero (cache all).

Keywords

None

Version History

5.2	Introduced

HDF_GR_SETCOMPRESS

This function specifies the type of compression for the specified HDF GR raster image.

Syntax

Result = HDF_GR_SETCOMPRESS(*ri_id*, *comp_type*, *comp_prm*)

Return Value

Returns SUCCEED (0) if successful or FAIL (-1) otherwise.

Arguments

ri_id

Raster image identifier returned by HDF_GR_CREATE or HDF_GR_SELECT.

comp_type

Compression method for the image data. Valid values are:

- 0 = no compression
- 1 = RLE run-length encoding
- 3 = Skipping Huffman compression
- 4 = GZIP compression
- 6 = JPEG compression

comp_prm

Compression parameters. If Skipping Huffman is used, set *comp_parm* to the skipping size (the size in bytes of the data elements). If GZIP compression is used, set *comp_parm* to an integer ranging from 1 (fastest) to 9 (most compressed).

Keywords

None

Version History

5.2	Introduced

HDF_GR_SETEXTERNALFILE

This function causes the specified HDF GR raster image be written to the specified external file, at the specified offset. Data can be moved only once for any given raster image, and it is the user's responsibility to make sure the external data file is kept with the “original” file. If the raster image already exists, its data will be moved to the external file. Space occupied by the data in the primary file will not be released. If the raster image does not exist, its data will be written to the external file during the subsequent calls to HDF_GR_WRITEDATA.

Syntax

Result = HDF_GR_SETEXTERNALFILE(*ri_id*, *filename*, *offset*)

Return Value

Returns SUCCEED (0) if successful or FAIL (-1) otherwise.

Arguments

ri_id

Raster image identifier returned by HDF_GR_CREATE or HDF_GR_SELECT.

filename

Name of the external file.

offset

Offset in bytes from the beginning of the external file to where the data will be written.

Keywords

None

Version History

5.2	Introduced

HDF_GR_START

This function initializes the HDF GR interface for the specified file. This function is used with the HDF_GR_END procedure to define the extent of the HDF GR interface session. As with the start routines in the other interfaces, HDF_GR_START initializes the internal interface structures needed for the remaining HDF_GR_ routines. Use the general purpose routines HDF_OPEN and HDF_CLOSE to manage file access. The HDF_GR_ routines will not open and close HDF files.

Note

Failure to use HDF_CLOSE properly may result in lost data or corrupted HDF files.

Syntax

Result = HDF_GR_START(*file_id*)

Return Value

Returns the HDF GR interface identifier if successful or FAIL (-1) otherwise.

Arguments

file_id

File identifier returned by HDF_OPEN.

Keywords

None

Version History

5.2	Introduced

HDF_GR_WRITEIMAGE

This function writes the subsample of the raster image data stored in the variable data to the specified raster image. The subsample is defined by the values of the parameters start, stride and edge. The array start specifies the starting location of the subsample to be written. Valid values of each element in the array start are 0 to the size of the corresponding raster image dimension - 1.

Note

The correspondence between elements in the array start and the raster image dimensions in the HDF GR interface is different from that in the HDF SD interface. The array stride specifies the writing pattern along each dimension. For example, if one of the elements of the array stride is 1, then every element along the corresponding dimension of the array data will be written. If one of the elements of the stride array is 2, then every other element along the corresponding dimension of the array data will be written, and so on. The correspondence between elements of the array stride and the dimensions of the array data is the same as described above for the array start. Each element of the array edges specifies the number of data elements to be written along the corresponding dimension. The correspondence between the elements of the array edges and the dimensions of the array data is the same as described above for the array start.

Syntax

Result = HDF_GR_WRITEIMAGE(*ri_id*, *data* [, EDGE=*array*]
[, INTERLACE={0 | 1 | 2}] [, START=*array*] [, STRIDE=*array*])

Return Value

Returns SUCCEED (0) if successful or FAIL (-1) otherwise.

Arguments

ri_id

Raster image identifier returned by HDF_GR_CREATE or HDF_GR_SELECT.

data

The image data to be written.

Keywords

EDGE

Array containing the number of data elements that will be written along each dimension. If not specified, all data will be written.

INTERLACE

Set this keyword to a scalar value to select the interlace mode of the input data. Valid values are:

- 0 = Pixel interlace
- 1 = Line interlace
- 2 = Component interlace

HDF_GR_WRITEIMAGE will write the data in the correct interlace mode the raster image is in.

START

Array containing the two-dimensional coordinate of the initial location for the write. If not specified, the write starts at the first pixel in each dimension (start=[0,0]).

STRIDE

Array containing the number of data locations the current location is to be moved forward before each write. If not specified, data is written contiguously (stride = [1,1]).

Note

See HDF_GR_READIMAGE for further description of the EDGE, START, and STRIDE keywords.

Version History

5.2	Introduced

HDF_GR_WRITELUT

This function writes a palette with the specified palette data and identifier. The palette data itself is stored in the *pal_data* variable. The data types supported by HDF are listed in "IDL and HDF Data Types" on page 317.

Syntax

Result = HDF_GR_WRITELUT(*pal_id*, *pal_data*)

Return Value

Returns SUCCEED (0) if successful or FAIL (-1) otherwise.

Arguments

pal_id

Palette identifier returned by HDF_GR_GETLUTID.

pal_data

Palette data to be written.

Keywords

None

Version History

5.2	Introduced

HDF_HDF2IDLTYPE

This function converts an HDF data type code into an IDL variable type code. See the IDL SIZE function and tables 3-2 through 3-4 in *Scientific Data Formats* for actual values.

Syntax

Result = HDF_HDF2IDLTYPE(*hdftypecode*)

Return Value

Returns the IDL variable type code (See SIZE). A return value of zero means the type could not be mapped.

Arguments

hdftypecode

An HDF data type code (long).

Keywords

None

Examples

```
PRINT, HDF_HDF2IDLTYPE( 6 )
```

Version History

5.2	Introduced

See Also

HDF_IDL2HDFTYPE

HDF_IDL2HDFTYPE

This function converts an IDL variable type code into an HDF data type code. See the IDL SIZE function and tables 3-2 through 3-4 in *Scientific Data Formats* for actual values.

Syntax

Result = HDF_IDL2HDFTYPE(*idltypecode*)

Return Value

Returns the HDF data type code. A return value of zero means the type could not be mapped.

Arguments

idltypecode

An IDL variable type code (long).

Keywords

None

Examples

```
iType = SIZE(5.0d,/TYPE)
PRINT, HDF_IDL2HDFTYPE( iType )
```

Version History

5.2	Introduced

See Also

HDF_HDF2IDLTYPE

HDF_ISHDF

The HDF_ISHDF function determines whether or not a specified file in an HDF file.

Warning

This routine bases its judgement as to whether or not a file is an HDF file on the first few bytes of the file. Therefore, it is possible that HDF_ISHDF will identify the file as an HDF file, but HDF_OPEN will not be able to open the file (because it is corrupted).

Syntax

Result = HDF_ISHDF(*Filename*)

Return Value

Returns true (1) if the file is an HDF file and false (0) if the file either is not an HDF file or does not exist.

Arguments

Filename

A scalar string containing the name of the file to be tested.

Keywords

None

Version History

Pre 4.0	Introduced

HDF_LIB_INFO

The HDF_LIB_INFO procedure returns information about the HDF Library being used by this version of IDL, or information about the version of HDF used to create a particular HDF file.

Syntax

HDF_LIB_INFO, [*FileHandle*] [, MAJOR=*variable*] [, MINOR=*variable*]
[, RELEASE=*variable*] [, VERSION=*variable*]

Arguments

FileHandle

The HDF filehandle returned from a previous call to HDF_OPEN.

Keywords

MAJOR

Set this keyword equal to a named variable that will contain the major version number of the HDF library currently in use by IDL. If the *FileHandle* argument is supplied, the variable will contain the major version number of the HDF library used by that particular HDF file.

MINOR

Set this keyword equal to a named variable that will contain the minor version number of the HDF library currently in use by IDL. If the *FileHandle* argument is supplied, the variable will contain the minor version number of the HDF library used by that particular HDF file.

RELEASE

Set this keyword equal to a named variable that will contain the release number of the HDF library currently in use by IDL. If the *FileHandle* argument is supplied, the variable will contain the release number of the HDF library used by that particular HDF file.

VERSION

Set this keyword equal to a named variable that will contain the version number text string of the HDF library currently in use by IDL. If the *FileHandle* argument is supplied, the variable will contain the version number text string of the HDF library used by that particular HDF file.

Examples

Example 1

```
HDF_LIB_INFO, MAJOR=MAJOR, MINOR=MINOR, VERSION=VER, RELEASE=REL
PRINT, 'IDL ', !version.release, ' uses HDF Library ', $
   MAJOR, MINOR, REL, FORMAT='(A,A,A,I1,".",I1,"r",I1,A)'
PRINT, VER
```

IDL Output

```
IDL 5.3 uses HDF Library 4.1r3
NCSA HDF Version 4.1 Release 3, May 1999
```

Example 2

The following example tests the version of HDF used to create a particular file. Note that the strings returned will depend solely upon the version of the HDF library used to create the file. In this example, it is the same as the library compiled into the current version of IDL since it is the current IDL that is creating the file.

```
file='example.hdf'
id=HDF_OPEN(file, /CREATE)
HDF_LIB_INFO, id, VERSION=VER
PRINT, 'The file ', file,' was created with : ', VER
HDF_CLOSE, id
```

IDL Output

```
The file example.hdf was created with :
NCSA HDF Version 4.1 Release 3, May 1999
```

Version History

5.1	Introduced

HDF_NEWREF

The HDF_NEWREF function returns the next available reference number for an HDF file.

Syntax

Result = HDF_NEWREF(*FileHandle*)

Return Value

Returns the next available reference number.

Arguments

FileHandle

The HDF file handle returned from a previous call to HDF_OPEN.

Keywords

None

Version History

Pre 4.0	Introduced

HDF_NUMBER

The HDF_NUMBER function returns the number of tags in an HDF file or the number of references associated with a given tag.

Syntax

Result = HDF_NUMBER(*FileHandle* [, TAG=*integer*])

Return Value

Returns either the number of tags in the file or the number of references associated with the specified tag.

Arguments

FileHandle

The HDF file handle returned from a previous call to HDF_OPEN.

Keywords

TAG

Set this keyword to an integer tag number or the string `'*'`. If this keyword is set to a tag number, HDF_NUMBER returns the number of references associated with the given tag. If this keyword is set to the string `'*'`, or is not specified, HDF_NUMBER returns the total number of tags in the HDF file.

Version History

Pre 4.0	Introduced

HDF_OPEN

The HDF_OPEN function opens or creates an HDF file for reading and/or writing.

Note that any combination of the READ, WRITE and CREATE keywords is valid.

Syntax

Result = HDF_OPEN(*Filename* [, /ALL] [, /CREATE] [, NUM_DD=*value*]
[, /RDWR] [, /READ] [, /WRITE])

Return Value

If successful, a non-zero file handle (a longword integer) is returned. Longword -1 is returned on failure.

Arguments

Filename

A scalar string containing the name of the file to be opened.

Keywords

ALL

Set this keyword to create a new HDF file with read and write access. Setting this keyword is equivalent to:

```
HDF_OPEN(filename, /READ, /WRITE, /CREATE)
```

CREATE

Set this keyword to create a new HDF file.

NUM_DD

Use this keyword to override the machine default for the number of data descriptors to be allocated per DD block. For example:

```
H = HDF_OPEN('foo.hdf',/CREATE,/WRITE, NUM_DD=100)
```

RDWR

Set this keyword to open file with both read and write access. Setting this keyword is equivalent to:

```
HDF_OPEN(filename, /READ, /WRITE)
```

READ

Set this keyword to open the file with read access.

WRITE

Set this keyword to open the file with write access.

Version History

Pre 4.0	Introduced

HDF_PACKDATA

This function packs a set IDL variable into an array of raw byte data. It is useful in constructing the input to multi-field HDF Vdata writing routines, such as those found in HDF-EOS, from a set of IDL variables. The packed data is output as an array of bytes which is organized as a number of records. Each record consists of one or more data fields. A record is defined using the HDF_TYPE and HDF_ORDER keywords. These define the record layout in terms of HDF data types. This function first converts the input arrays into the type defined by the HDF_TYPE keyword using IDL type conversion rules. The function then walks through the input IDL arrays and copies the values into output array. There must be as many entries in the HDF_TYPE and HDF_ORDER keywords as there are data arguments. The function will output as many complete records as can be created from the input data arrays or the value of the NREC keyword, whichever is smaller.

Syntax

Result = HDF_PACKDATA(*data1* [, *data2* [, *data3* [, *data4* [, *data5* [, *data6* [, *data7* [, *data8*]]]]]]] [, HDF_ORDER=*array*] [, HDF_TYPE=*array*] [, NREC=*records*])

Return Value

Returns a 2-D BYTE array of packed data. The trailing dimension corresponds to each record in the input data.

Arguments

data1...data8

These arguments specify IDL arrays to be packed. The arguments are first converted to the types specified by HDF_TYPE. If the corresponding HDF_ORDER value is greater than one, more than one value will be read from the input array and placed in the packed array for each record. Strings are output as fixed width fields. If an input string is longer than its HDF_ORDER value, it is truncated before being packed. If an input string is shorter than its HDF_ORDER value, the extra space is filled with the value 0.

Keywords

HDF_ORDER

Set this keyword to an array the same length as the number of data fields. The value in the array is equal to the number of elements in the data argument for each record. In the case of strings, this is the length (in characters) of the string to be packed. A value of zero is interpreted as one element. The default for this keyword is an array of ones.

HDF_TYPE

Set this keyword to an array the same length as the number of data fields. The value in the array is an HDF data type for each argument. The IDL variables are converted to these types before being packed into the output array. The default for this keyword is an array of the value 5 (an HDF 32 bit float). See "IDL and HDF Data Types" on page 317 for valid values.

NREC

Set this keyword to the number of records to be packed. The default is to pack as many complete records as can be formed by all of the input arrays.

Examples

See HDF_UNPACKDATA.

Version History

5.2	Introduced

See Also

HDF_UNPACKDATA, EOS_PT_WRITELEVEL, HDF_VD_WRITE

HDF_READ

See HDF_READ in the *IDL Reference Guide*.

HDF_SD_ADDDATA

The HDF_SD_ADDDATA procedure writes a hyperslab of values to an SD dataset. By default, the output data is transposed. This transposition puts the data in column order, which is more efficient in HDF than row order (which is more efficient in IDL). In the rare cases where it is necessary to write the data without transposing, set the NOREVERSE keyword. The OFFSET, COUNT, and STRIDE keywords are similarly affected by the NOREVERSE keyword.

Syntax

HDF_SD_ADDDATA, *SDdataset_id*, *Data* [, COUNT=*vector*] [, /NOREVERSE] [, START=*vector*] [, STRIDE=*vector*]

Arguments

SDdataset_id

An SD dataset ID as returned by HDF_SD_SELECT or HDF_SD_CREATE.

Data

The data to be written.

Keywords

COUNT

Set this keyword to a vector of counts (i.e., the number of items) to be written in each dimension. The default is to write all available data. Use caution when using this keyword. See the second example, below.

NOREVERSE

Set this keyword to prevent HDF_SD_ADDDATA's transposition of *Data* and any vectors specified by keywords into column order.

START

Set this keyword to a vector that contains the starting position for the data. The default position is [0, 0, ..., 0].

STRIDE

Set this keyword to a vector that contains the strides, or sampling intervals, between accessed values of the NetCDF variable. The default stride vector is that for a contiguous write: [0, 0, ..., 0].

Examples

The following example writes a 230-element by 380-element byte image to an SD dataset, then reads it back as a 70 by 100 image starting at (40, 20), sampling every other Y pixel and every third X pixel:

```
start = [40, 20] ; Set the start vector.
count = [70, 100] ; Set the count vector.
stride = [2, 3] ; Set the stride vector.
image = DIST(230, 380) ; Create the image.
TV, image ; Display the image.
; Create a new HDF file in SD mode:
SDinterface_id = HDF_SD_START('image.hdf', /CREATE)
; Define a new SD dataset:
SDdataset_id = HDF_SD_CREATE(SDinterface_id, 'image', [230, 380],
/BYTE)
HDF_SD_ADDDATA, SDdataset_id, image ; Write the image into the
dataset.
HDF_SD_GETDATA, SDdataset_id, full ; Retrieve the full image.
; Retrieve the sub-sampled image:
HDF_SD_GETDATA, SDdataset_id, small, COUNT=count, $
   START=start, STRIDE=stride
HDF_SD_ENDACCESS, SDdataset_id
HDF_SD_END, SDinterface_id
HELP, full, small ; Print information about the images.
ERASE ; Erase the window.
TV, full; Display the full image.
TV, small ; Display the sub-sampled image.
```

IDL prints:

```
FULL   BYTE = Array(230, 380)
SMALL  BYTE = Array(70, 100)
```

Continuing with our example, suppose we want to write the center 50 by 100 pixels of the image to the file. You might be tempted to try:

```
HDF_SD_ADDDATA, SDdataset_id, image, START=[90, 90],
COUNT=[50,100]
```

You will find, however, that this captures the lower left-hand corner of the original image, rather than the center. To write the data from the center, subset the original image, choosing the data from the center:

```
HDF_SD_ADDDATA, SDdataset_id, image(90:139, 90:189), START=[90,
90],$
   COUNT=[50,100] ; This is the correct way to add the data.
HDF_SD_ENDACCESS, SDdataset_id ; End SD access.
HDF_SD_END, SDinterface_id ; Close the file.
```

Version History

4.0	Introduced

See Also

HDF_SD_GETDATA

HDF_SD_ATTRFIND

The HDF_SD_ATTRFIND function locates the index of an HDF attribute given its name. The attribute can be global or from a specific dataset. If an attribute is located, its index is returned. Otherwise, -1 is returned. Once an attribute's index is known, the HDF_SD_ATTRINFO function can be used to read that attribute.

Syntax

Result = HDF_SD_ATTRFIND(*SD_id*, *Name*)

Arguments

SD_id

An SD interface ID as returned by HDF_SD_START (i.e., a global attribute's "SDinterface_id"), or HDF_SD_SELECT/HDF_SD_CREATE (i.e., a dataset's "SDdataset_id").

Name

A string containing the name of the attribute whose index is to be returned.

Keywords

None

Examples

```
; Open an HDF file and start the SD interface:
SDinterface_id = HDF_SD_START('demo.hdf')
; Find "TITLE", a global attribute:
gindex = HDF_SD_ATTRFIND(SDinterface_id, 'TITLE')
; Get the ID for the first dataset:
SDdataset_id = HDF_SD_SELECT(SDinterface_id, 1)
; Read attribute info:
HDF_SD_ATTRINFO,SDinterface_id,gindex, NAME=name, TYPE=type,
COUNT=count
; Print info about the returned variables:
HELP, type, count, name
; Find the "LOCATION" dataset attribute:
dindex = HDF_SD_ATTRFIND(SDdataset_id, 'LOCATION')
; Read attribute info:
```

```
HDF_SD_ATTRINFO,SDdataset_id,dindex,NAME=name,TYPE=type,COUNT=count
```

IDL Output

```
TYPE STRING = 'STRING'
COUNT LONG = 8
NAME STRING = 'TITLE'
```

Version History

4.0	Introduced

See Also

HDF_SD_ATTRINFO, HDF_SD_ATTRSET, HDF_SD_SELECT

HDF_SD_ATTRINFO

The HDF_SD_ATTRINFO procedure reads or retrieves information about an SD attribute. The attribute can be global or from a specific dataset. If an attribute is not present, an error message is printed.

Syntax

HDF_SD_ATTRINFO, *SD_id*, *Attr_Index* [, COUNT=*variable*] [, DATA=*variable*]
[, HDF_TYPE=*variable*] [, NAME=*variable*] [, TYPE=*variable*]

Arguments

SD_id

An SD interface ID as returned by HDF_SD_START (i.e., a global attribute's "SDinterface_id"), or HDF_SD_SELECT/HDF_SD_CREATE (i.e., a dataset's "SDdataset_id").

Attr_Index

The attribute index, can either be obtained by calling HDF_SD_ATTRFIND if a particular attribute name is known or can be obtained with a 0-based index sequentially referencing the attribute.

Keywords

COUNT

Set this keyword to a named variable in which the total number of values in the specified attribute is returned.

DATA

Set this keyword to a named variable in which the attribute data is returned.

HDF_TYPE

Set this keyword to a named variable in which the HDF type of the attribute is returned as a scalar string. Possible returned values are DFNT_NONE, DFNT_CHAR8, DFNT_FLOAT32, DFNT_FLOAT64, DFNT_INT8, DFNT_INT16, DFNT_INT32, DFNT_UINT8, DFNT_UINT16, and DFNT_UINT32. (See "IDL and HDF Data Types" on page 317 for valid values.)

NAME

Set this keyword to a named variable in which the name of the attribute is returned.

TYPE

Set this keyword to a named variable in which the IDL type of the attribute is returned as a scalar string. Possible returned values are BYTE, INT, LONG, FLOAT, DOUBLE, STRING, or UNKNOWN.

Examples

```
; Open an HDF file and start the SD interface:
SDinterface_id = HDF_SD_START('demo.hdf')
; Find a global attribute:
gindex = HDF_SD_ATTRFIND(SDinterface_id, 'TITLE')
; Retrieve attribute info:
HDF_SD_ATTRINFO, SDinterface_id, gindex, NAME=n, TYPE=t, $
   COUNT=c, DATA=d, HDF_TYPE=h
; Print information about the returned variables:
HELP, n, t, c, h
; Return the SD dataset ID for the first dataset (index 0):
SDdataset_id = HDF_SD_SELECT(SDinterface_id, 0)
; Find a dataset attribute:
dindex = HDF_SD_ATTRFIND(SDdataset_id, 'LOCATION')
; Retrieve attribute info:
HDF_SD_ATTRINFO,SDdataset_id, dindex, NAME=n, TYPE=t, $
   COUNT=c, DATA=d
; Print information about the new returned variables:
HELP, n, t, c, d
```

IDL Output

```
N STRING = 'TITLE'
T STRING = 'STRING'
C LONG = 17
D STRING = '5th Ave Surf Shop'
H STRING = 'DFNT_CHAR8'

N STRING = 'LOCATION'
T STRING = 'STRING'
C LONG = 15
D STRING = 'MELBOURNE BEACH'
```

Version History

4.0	Introduced

See Also

HDF_SD_ATTRFIND, HDF_SD_ATTRSET, HDF_SD_CREATE, HDF_SD_SELECT, HDF_SD_START

HDF_SD_ATTRSET

The HDF_SD_ATTRSET procedure writes attributes to an open HDF SD dataset. If no data type is specified, the data type is taken from the *Values* argument.

Syntax

HDF_SD_ATTRSET, *SD_id*, *Attr_Name*, *Values* [, *Count*] [, /BYTE] [, /DFNT_CHAR] [, /DFNT_FLOAT32] [, /DFNT_FLOAT64] [, /DFNT_INT8] [, /DFNT_INT16] [, /DFNT_INT32] [, /DFNT_UINT8] [, /DFNT_UINT16] [, /DFNT_UINT32] [, /DOUBLE] [, /FLOAT] [, /INT] [, /LONG] [, /SHORT] [, /STRING]

Arguments

SD_id

An SD interface ID as returned by HDF_SD_START (i.e., a global attribute's "SDinterface_id"), or HDF_SD_SELECT/HDF_SD_CREATE (i.e., a dataset's "SDdataset_id").

Attr_Name

A string containing the name of the attribute to be written.

Values

The attribute values to be written.

Count

An optional integer argument specifying how many items are to be written. Count must be less than or equal to the number of elements in the Values argument.

Keywords

BYTE

Set this keyword to indicate that the attribute is composed of bytes. Data will be stored with the HDF DFNT_UINT8 data type. Setting this keyword is the same as setting the DFNT_UINT8 keyword.

DFNT_CHAR

Set this keyword to create an attribute of HDF type DFNT_CHAR. Setting this keyword is the same as setting the STRING keyword.

DFNT_FLOAT32

Set this keyword to create an attribute of HDF type DFNT_FLOAT32. Setting this keyword is the same as setting the FLOAT keyword.

DFNT_FLOAT64

Set this keyword to create an attribute of HDF type DFNT_FLOAT64. Setting this keyword is the same as setting the DOUBLE keyword.

DFNT_INT8

Set this keyword to create an attribute of HDF type DFNT_INT8.

DFNT_INT16

Set this keyword to create an attribute of HDF type DFNT_INT16. Setting this keyword is the same as setting either the INT keyword or the SHORT keyword.

DFNT_INT32

Set this keyword to create an attribute of HDF type DFNT_INT32. Setting this keyword is the same as setting the LONG keyword.

DFNT_UINT8

Set this keyword to create an attribute of HDF type DFNT_UINT8. Setting this keyword is the same as setting the BYTE keyword.

DFNT_UINT16

Set this keyword to create an attribute of HDF type DFNT_UINT16.

DFNT_UINT32

Set this keyword to create an attribute of HDF type DFNT_UINT32.

DOUBLE

Set this keyword to indicate that the attribute is composed of double-precision floating-point values. Data will be stored with the HDF DFNT_FLOAT64 data type. Setting this keyword is the same as setting the DFNT_FLOAT64 keyword.

FLOAT

Set this keyword to indicate that the attribute is composed of single-precision floating-point values. Data will be stored with the HDF DFNT_FLOAT32 data type. Setting this keyword is the same as setting the DFNT_FLOAT32 keyword.

INT

Set this keyword to indicate that the attribute is composed of 2-byte integers. Data will be stored with the HDF DFNT_INT16 data type. Setting this keyword is the same as setting either the SHORT keyword or the DFNT_INT16 keyword.

LONG

Set this keyword to indicate that the attribute is composed of longword integers. Data will be stored with the HDF DFNT_INT32 data type. Setting this keyword is the same as setting the DFNT_INT32 keyword.

SHORT

Set this keyword to indicate that the attribute is composed of 2-byte integers. Data will be stored with the HDF DFNT_INT16 data type. Setting this keyword is the same as setting either the INT keyword or the DFNT_INT16 keyword.

STRING

Set this keyword to indicate that the attribute is composed of strings. Data will be stored with the HDF DFNT_CHAR8 data type. Setting this keyword is the same as setting the DFNT_CHAR8 keyword.

Examples

```
fid = HDF_OPEN('demo.hdf', /ALL) ; Create a new HDF file.
SDinterface_id = HDF_SD_START('demo.hdf', /RDWR) ; Start the SD
interface.
; Create a global attribute:
HDF_SD_ATTRSET, SDinterface_id, 'TITLE', 'MY TITLE GLOBAL', 16
; Create another global attribute:
HDF_SD_ATTRSET, SDinterface_id, 'RANGE', [-99.88,55544.2], /DOUBLE
```

```
; Create a dataset:
SDdataset_id = HDF_SD_CREATE(SDinterface_id, 'var1', [10,20],
/FLOAT)
; Add a dataset attribute:
HDF_SD_ATTRSET, SDdataset_id, 'TITLE', 'MY TITLE SDinterface_id',
15
; Find the recently-created RANGE attribute:
index=HDF_SD_ATTRFIND(SDinterface_id, 'RANGE')
; Retrieve data from RANGE:
HDF_SD_ATTRINFO,SDinterface_id,index,NAME=atn,COUNT=atc,TYPE=att,D
ATA=d
; Print information about the returned variables:
HELP, atn, atc, att
; Print the data returned in variable d with the given format:
PRINT, d, FORMAT='(F8.2,x,F8.2)'
HDF_SD_ENDACCESS, SDdataset_id ; End access to the HDF file.
HDF_SD_END, SDinterface_id
HDF_CLOSE, fid
```

IDL Output

```
ATN STRING = 'RANGE'
ATC LONG = 2
ATT STRING = 'DOUBLE'

-99.88 55544.20
```

Version History

4.0	Introduced

See Also

HDF_SD_ATTRFIND, HDF_SD_ATTRINFO, HDF_SD_CREATE, HDF_SD_SELECT

HDF_SD_CREATE

The HDF_SD_CREATE function creates and defines a Scientific Dataset (SD) for an HDF file. Keywords can be set to specify the data type. If no keywords are present a floating-point dataset is created.

Syntax

Result = HDF_SD_CREATE(*SDinterface_id*, *Name*, *Dims* [, /BYTE] [, /DFNT_CHAR8] [, /DFNT_FLOAT32] [, /DFNT_FLOAT64] [, /DFNT_INT8] [, /DFNT_INT16] [, /DFNT_INT32] [, /DFNT_UINT8] [, /DFNT_UINT16] [, /DFNT_UINT32] [, /DOUBLE] [, /FLOAT] [, HDF_TYPE=*type*] [, /INT] [, /LONG] [, /SHORT] [, /STRING])

Return Value

The returned value of this function is the SDS ID of the newly-created dataset.

Arguments

SDinterface_id

An SD ID as returned by HDF_SD_START.

Name

A string containing the name of the variable to be created.

Dims

A 1-based vector specifying the dimensions of the variable. If an UNLIMITED dimension is desired, set the last vector element to zero or a negative number.

Keywords

BYTE

Set this keyword to indicate that the dataset is composed of bytes. Data will be stored with the HDF DFNT_UINT8 data type. Setting this keyword is the same as setting the DFNT_UINT8 keyword.

DFNT_CHAR8

Set this keyword to create a data set of HDF type DFNT_CHAR8. Setting this keyword is the same as setting the STRING keyword.

DFNT_FLOAT32

Set this keyword to create a data set of HDF type DFNT_FLOAT32. Setting this keyword is the same as setting the FLOAT keyword.

DFNT_FLOAT64

Set this keyword to create a data set of HDF type DFNT_FLOAT64. Setting this keyword is the same as setting the DOUBLE keyword.

DFNT_INT8

Set this keyword to create a data set of HDF type DFNT_INT8.

DFNT_INT16

Set this keyword to create a data set of HDF type DFNT_INT16. Setting this keyword is the same as setting either the INT keyword or the SHORT keyword.

DFNT_INT32

Set this keyword to create a data set of HDF type DFNT_INT32. Setting this keyword is the same as setting the LONG keyword.

DFNT_UINT8

Set this keyword to create a data set of HDF type DFNT_UINT8. Setting this keyword is the same as setting the BYTE keyword.

DFNT_UINT16

Set this keyword to create a data set of HDF type DFNT_UINT16.

DFNT_UINT32

Set this keyword to create a data set of HDF type DFNT_UINT32.

DOUBLE

Set this keyword to indicate that the dataset is composed of double-precision floating-point values. Data will be stored with the HDF DFNT_FLOAT64 data type. Setting this keyword is the same as setting the DFNT_FLOAT64 keyword.

FLOAT

Set this keyword to indicate that the dataset is composed of single-precision floating-point values. Data will be stored with the HDF DFNT_FLOAT32 data type. Setting this keyword is the same as setting the DFNT_FLOAT32 keyword.

HDF_TYPE

Set this keyword to the type of data set to create. Valid values are: DFNT_CHAR8, DFNT_FLOAT32, DFNT_FLOAT64, DFNT_INT8, DFNT_INT16, DFNT_INT32, DFNT_UINT8, DFNT_UINT16, DFNT_UINT32. (See "IDL and HDF Data Types" on page 317 for valid values.)

For example:

```
type = HDF_IDL2HDFTYPE(SIZE(myData, /type))
SDdataset_id = HDF_SD_CREATE(f_id, "name", dims, HDF_TYPE=type)
```

INT

Set this keyword to indicate that the dataset is composed of 2-byte integers. Data will be stored with the HDF DFNT_INT16 data type. Setting this keyword is the same as setting either the SHORT keyword or the DFNT_INT16 keyword.

LONG

Set this keyword to indicate that the dataset is composed of longword integers. Data will be stored with the HDF DFNT_INT32 data type. Setting this keyword is the same as setting the DFNT_INT32 keyword.

SHORT

Set this keyword to indicate that the dataset is composed of 2-byte integers. Data will be stored with the HDF DFNT_INT16 data type. Setting this keyword is the same as setting either the INT keyword or the DFNT_INT16 keyword.

STRING

Set this keyword to indicate that the dataset is composed of strings. Data will be stored with the HDF DFNT_CHAR8 data type. Setting this keyword is the same as setting the DFNT_CHAR8 keyword.

Examples

```
; Create a new HDF file:
SDinterface_id = HDF_SD_START('test.hdf', /CREATE)
; Create an dataset that includes an unlimited dimension:
SDdataset_id = HDF_SD_CREATE(SDinterface_id, 'var1', [9,40,0],
/SHORT)
```

The example for HDF_SD_ATTRSET also demonstrates the use of this routine.

Version History

4.0	Introduced

See Also

HDF_OPEN, HDF_SD_ENDACCESS, HDF_SD_SELECT

HDF_SD_DIMGET

The HDF_SD_DIMGET procedure retrieves information about an SD dataset dimension.

Syntax

HDF_SD_DIMGET, *Dim_ID* [, /COUNT] [, COMPATIBILITY=*variable*]
[, /FORMAT] [, /LABEL] [, /NAME] [, /NATTR] [, /SCALE] [, /TYPE]
[, /UNIT]

Arguments

Dim_ID

A dimension ID as returned by HDF_SD _DIMGETID.

Keywords

COUNT

Set this keyword to return the dimension size.

COMPATIBILITY

Set this keyword to a named variable that will contain a string indicating the dimensional compatibility of the current dimension. Possible values are "BW_COMP" (backwards compatible), "BW_INCOMP" (backwards incompatible), or "FAIL" (the information is unavailable). For further information about dimensional compatibilities, see the HDF User's Guide, and the BW_INCOMP keyword of HDF_SD_DIMSET. By default, IDL writes HDF files in "BW_COMP" mode.

FORMAT

Set this keyword to return the dimension format description string.

LABEL

Set this keyword to return the dimension label description string.

NAME

Set this keyword to return the dimension name.

NATTR

Set this keyword to return the number of attributes for the dimension.

SCALE

Set this keyword to return the scale of the dimension.

TYPE

Set this keyword to return a string describing the data’s type (i.e., ‘BYTE’).

UNIT

Set this keyword to return the dimension unit description string.

Examples

For an example using this routine, see the example for HDF_SD_DIMSET.

Version History

4.0	Introduced

See Also

HDF_SD_CREATE, HDF_SD_DIMGETID, HDF_SD_DIMSET,
HDF_SD_SELECT

HDF_SD_DIMGETID

The HDF_SD_DIMGETID function returns a dimension ID given a dataset’s “SDdataset_id” and a dimension number.

Syntax

Result = HDF_SD_DIMGETID(*SDdataset_id*, *Dimension_Number*)

Return Value

Returns the dimension identifier.

Arguments

SDdataset_id

An SD dataset ID as returned by HDF_SD _SELECT or HDF_SD_CREATE.

Dimension_Number

A zero-based dimension number. The dimension number must be greater than or equal to 0 and less than the maximum dimension number, or *rank*.

Keywords

None

Examples

For an example illustrating this routine, see the documentation for HDF_SD_DIMSET.

Version History

4.0	Introduced

See Also

HDF_SD_CREATE, HDF_SD_DIMGET, HDF_SD_DIMSET, HDF_SD_SELECT

HDF_SD_DIMSET

The HDF_SD_DIMSET procedure sets the scale and data strings for an SD dimension.

Syntax

HDF_SD_DIMSET, *Dim_ID* [, /BW_INCOMP] [, FORMAT=*string*]
[, LABEL=*string*] [, NAME=*string*] [, SCALE=*vector*] [, UNIT=*string*]

Arguments

Dim_ID

A dimension ID as returned by HDF_SD _DIMGETID.

Keywords

BW_INCOMP

Set this keyword to write SD dimensions in the “new” (HDF4.1 and later) style. Versions of HDF prior to HDF 4.0 beta 2 were inefficient in the use of SD dimensions. HDF now uses a new internal representation of SD dimensions. If the BW_INCOMP keyword is not set, or is explicitly set equal to zero, the current version of HDF writes SD dimensions in both the pre-HDF 4.0 format AND the “new” format. This default behavior is called the BW_COMP dimensional compatibility representation.

Setting the BW_INCOMP keyword causes the current dimension to be written in only the “new” (HDF4.1 and later) format. Depending on your HDF file, using this new format can reduce the size of the HDF by up to a factor of 2, but at the expense of incompatibility with pre HDF 4.0 beta 2 applications (IDL version 4, for example). The COMPATIBILITY keyword of HDF_SD_DIMGET can be used to check the dimensional compatibility of an HDF dimension.

Note

Future versions of HDF will recognize *only* the “new” (BW_INCOMP) dimensional representation.

FORMAT

A string for the dimension format.

LABEL

A string for the dimension label.

NAME

A string for the dimension name.

SCALE

A vector containing the dimension scale values.

UNIT

A string for the dimension unit.

Examples

```
; Initialize the SD interface:
SDinterface_id = HDF_SD_START('myhdf.hdf', /RDWR)
; Create 3 dimensions:
SDdataset_id = HDF_SD_CREATE(SDinterface_id, 'var1', [10,20,0],
/LONG)
; Select the first dimension:
dim_id=HDF_SD_DIMGETID(SDdataset_id,0)
; Set the data strings and scale for the first dimension:
HDF_SD_DIMSET, dim_id, NAME='d1', LABEL='l1', $
   FORMAT='f1', UNIT='u1', SCALE=FINDGEN(10)
HDF_SD_ENDACCESS, SDdataset_id
; Close the HDF file to ensure everything is written:
HDF_SD_END, SDinterface_id
; Reopen the file:
SDinterface_id = HDF_SD_START('myhdf.hdf')
; Select the first dimension:
dim_id = HDF_SD_DIMGETID(SDdataset_id,0)
; Retrieve the information:
HDF_SD_DIMGET, dim_id, NAME=d1, LABEL=l1, FORMAT=f1, $
   UNIT=u1, SCALE=sc, COUNT=cnt, NATTR=natt, TYPE=type
; Print information about the returned variables:
HELP, d1, l1, f1, u1, sc, cnt, natt, type
; Close the SD interface:
HDF_SD_ENDACCESS,SDdataset_id
HDF_SD_END, SDinterface_id
```

IDL Output

```
D1 STRING = 'd1'
L1 STRING = 'l1'
F1 STRING = 'f1'
U1 STRING = 'u1'
SC FLOAT = Array(10)
CNT LONG = 10
NATT LONG = 3
TYPE STRING = 'FLOAT'
```

Version History

4.0	Introduced

See Also

HDF_SD_CREATE, HDF_SD_DIMGET, HDF_SD_DIMGETID, HDF_SD_SELECT

HDF_SD_END

The HDF_SD_END procedure closes the SD interface to an HDF file. Failure to close the file without a call to HDF_SD_END results in the loss of any changed or added SD data. Therefore, HDF_SD_END calls should always be paired with calls to HDF_SD_START. Before HDF_SD_END is called, all access to SD datasets should be terminated with calls to HDF_SD_ENDACCESS.

Syntax

HDF_SD_END, *SDinterface_id*

Arguments

SDinterface_id

An SD interface ID as returned by HDF _SD_START.

Keywords

None

Examples

```
; Open a new HDF file:
SDinterface_id = HDF_SD_START('test.hdf', /CREATE)
; Various commands could now be used to access SD data
; in the HDF file.
; When done with datasets, access should be ended with
; calls to HDF_SD_ENDACCESS:
HDF_SD_ENDACCESS, SDdataset_id_1
; When done with an HDF file, it should be closed:
HDF_SD_END, SDinterface_id
```

Another example can be seen in the documentation for HDF_SD_ATTRSET.

Version History

4.0	Introduced

See Also

HDF_CLOSE, HDF_OPEN, HDF_SD_ENDACCESS, HDF_SD_START

HDF_SD_ENDACCESS

The HDF_SD_ENDACCESS procedure closes an SD dataset interface. Failure to close the interface can result in the loss of any changed or added SD data. This routine should be called once for each call to HDF_SD_START or HDF_SD_CREATE. After all SD dataset interfaces are closed, the HDF file can safely be closed with HDF_SD_END.

Syntax

HDF_SD_ENDACCESS, *SDinterface_id*

Arguments

SDdataset_id

An SD dataset ID as returned by HDF_SD_SELECT, or HDF_SD_CREATE.

Keywords

None

Examples

```
; Open a new HDF file:
SDinterface_id = HDF_SD_START('test.hdf', /CREATE)
; Access the HDF file:
SDdataset_id_1 = HDF_SD_SELECT(SDinterface_id,0)
; End access to any SD IDs:
HDF_SD_ENDACCESS, SDdataset_id_1
; Close the HDF file:
HDF_SD_END, SDinterface_id
```

Also see the example in HDF_SD_ATTRSET.

Version History

4.0	Introduced

See Also

HDF_CLOSE, HDF_OPEN, HDF_SD_CREATE, HDF_SD_ENDACCESS, HDF_SD_START

HDF_SD_FILEINFO

The HDF_SD_FILEINFO procedure retrieves the number of datasets and global attributes in an HDF file.

Syntax

HDF_SD_FILEINFO, *SDinterface_id*, *Datasets*, *Attributes*

Arguments

SDinterface_id

An SD interface ID as returned by HDF_SD_START.

Datasets

A named variable in which the total number of SD-type objects (i.e., the number of datasets + the number of dimensions) in the file is returned.

Attributes

A named variable in which the number of global attributes in the file is returned.

Keywords

None

Examples

```
; Start the SD interface:
SDinterface_id = HDF_SD_START('demo.hdf', /CREATE)
; Set a global attribute:
HDF_SD_ATTRSET,SDinterface_id, 'TITLE', 'MYTITLE'
; Set another one:
HDF_SD_ATTRSET,SDinterface_id, 'TITLE2', 'MYTITLE2'
; Create a dataset:
SDdataset_id = HDF_SD_CREATE(SDinterface_id, 'var1', [10,3])
; Retrieve info about the dataset:
HDF_SD_FILEINFO, SDinterface_id, datasets, attributes
; Print information about the returned variables:
HELP, datasets, attributes
; End SD access:
HDF_SD_ENDACCESS, SDdataset_id
```

```
; Close the SD interface:
HDF_SD_END, SDinterface_id
```

IDL Output

```
DATASETS LONG = 1
ATTRIBUTES LONG = 2
```

Version History

4.0	Introduced

See Also

HDF_SD_ATTRFIND, HDF_SD_ATTRINFO, HDF_SD_START

HDF_SD_GETDATA

The HDF_SD_GETDATA procedure retrieves a hyperslab of values from an SD dataset. By default, the retrieved data is transposed from HDF's column order format into IDL's row order which is more efficient in IDL. To retrieve the dataset without this transposition, set the NOREVERSE keyword.

Syntax

HDF_SD_GETDATA, *SDdataset_id*, *Data* [, COUNT=*vector*] [, /NOREVERSE] [, START=*vector*] [, STRIDE=*vector*]

Arguments

SDdataset_id

An SD dataset ID as returned by HDF_SD _SELECT or HDF_SD_CREATE.

Data

A named variable in which the values are returned.

Keywords

COUNT

Set this keyword to a vector containing the counts, or number of items, to be read. The default is to read all available data.

NOREVERSE

Set the keyword to retrieve the data without transposing the data from column to row order.

START

Set this keyword to a vector containing the starting position for the read. The default start position is [0, 0, ..., 0].

STRIDE

Set this keyword to a vector containing the strides, or sampling intervals, between accessed values of the HDF variable. The default stride vector is that for a contiguous read: [0, 0, ..., 0].

Examples

For an example using this routine, see the documentation for HDF_SD_ADDDATA.

Version History

4.0	Introduced

See Also

HDF_SD_ADDDATA, HDF_SD_GETINFO

HDF_SD_GETINFO

The HDF_SD_GETINFO procedure retrieves information about an SD dataset.

Warning

Reading a label, unit, format, or coordinate system string that has more than 256 characters can have unpredictable results.

Syntax

HDF_SD_GETINFO, *SDdataset_id* [, CALDATA=*variable*]
[, COORDSYS=*variable*] [, DIMS=*variable*] [, FILL=*variable*]
[, FORMAT=*variable*] [, HDF_TYPE=*variable*] [, LABEL=*variable*]
[, NAME=*variable]* [, NATTS=*variable*] [, NDIMS=*variable*] [, /NOREVERSE]
[, RANGE=*variable*] [, TYPE=*variable*] [, UNIT=*variable*]

Arguments

SDdataset_id

An SD dataset ID as returned by HDF_SD _SELECT or HDF_SD_CREATE.

Keywords

CALDATA

Set this keyword to a named variable in which the calibration data associated with the SD dataset is returned. The data is returned in a structure of the form:

For more information about calibration data, see the documentation for HDF_SD_SETINFO.

COORDSYS

Set this keyword to a named variable in which the coordinate system description string is returned.

DIMS

Set this keyword to a named variable in which the dimensions of the SD dataset are returned. For efficiency, these dimensions are returned in reverse order from their HDF format unless the NOREVERSE keyword is also set.

FILL

Set this keyword to a named variable in which the fill value of the SD dataset is returned. Note that a fill value *must* be set in the SD dataset. If a fill value is not set, the value of the variable named by this keyword will be undefined, and IDL will issue a warning message.

FORMAT

Set this keyword to a named variable in which the format description string is returned. If the format description string is not present, this variable will contain an empty string.

HDF_TYPE

Set this keyword to a named variable in which the HDF type of the SD dataset is returned as a scalar string. Possible returned values are DFNT_NONE, DFNT_CHAR8, DFNT_FLOAT32, DFNT_FLOAT64, DFNT_INT8, DFNT_INT16, DFNT_INT32, DFNT_UINT8, DFNT_UINT16, and DFNT_UINT32. (See "IDL and HDF Data Types" on page 317 for valid values.)

LABEL

Set this keyword to a named variable in which the label description string is returned. If the label description string is not present, this variable will contain an empty string.

NAME

Set this keyword to a named variable in which the SD dataset name is returned. If the SD dataset name is not present, this variable will contain an empty string.

NATTS

Set this keyword to a named variable in which the number of "NetCDF-style" attributes for the SD dataset is returned.

NDIMS

Set this keyword to a named variable in which the number of dimensions in the dataset is returned.

NOREVERSE

Set this keyword in conjunction with DIMS to return the variable dimensions in non-reversed form. By default, IDL reverses data and dimensions from the HDF format to improve efficiency.

RANGE

Set this keyword to a named variable in which the maximum and minimum of the current SD dataset is returned as a two-element vector. Note that a range *must* be set in the SD dataset. If the range is not set, the value of the variable named by this keyword will be undefined, and IDL will issue a warning message.

TYPE

Set this keyword to a named variable in which the IDL type of the SD dataset is returned as a scalar string. Possible returned values are BYTE, INT, LONG, FLOAT, DOUBLE, STRING, or UNKNOWN.

UNIT

Set this keyword to a named variable in which the unit description string is returned. If the unit description string is not present, this variable will contain an empty string.

Examples

For an example using this routine, see the documentation for HDF_SD_SETINFO.

Version History

4.0	Introduced

See Also

HDF_OPEN, HDF_SD_END, HDF_SD_SETINFO, HDF_SD_START

HDF_SD_IDTOREF

The HDF_SD_IDTOREF function converts a SD data set ID into a SD data set reference number. The reference number can be used to add the SD data set to a Vgroup through the HDF_VG interface. The tag number for an SD is 720.

Syntax

Result = HDF_SD_IDTOREF(*SDdataset_id*)

Return Value

Returns the SD data set reference number.

Arguments

SDdataset_id

A SDdataset_id as returned from HDF_SD_CREATE or HDF_SD_SELECT.

Keywords

None

Examples

```
; Create an SD data set and get the Reference number:
file_id = HDF_OPEN('demo.hdf', /ALL)
SDinterface_id = HDF_SD_START('demo.hdf', /RDWR)
dim=[100]
SDdataset_id = HDF_SD_CREATE(SDinterface_id, 'demo_data', dim,
/FLOAT)
ref = HDF_SD_IDTOREF(SDdataset_id)
HDF_SD_ADDDATA, SDdataset_id, FINDGEN(100)/10.45 + 2.98
HDF_SD_ENDACCESS, SDdataset_id
HDF_SD_END, SDinterface_id

; Use the Reference number to add the SD to a Vgroup:
SD_TAG = 720
vgID = HDF_VG_GETID(file_id,-1)
vg_handle = HDF_VG_ATTACH(file_id, vgID, /WRITE)
HDF_VG_SETINFO, vg_handle, name='data1', class='demo'
HDF_VG_ADDTR, vg_handle, SD_TAG, ref
```

```
; Use HDF_VG_INQTR to verify the SD was added correctly:
IF HDF_VG_INQTR(vg_handle, SD_TAG, ref) THEN $
   PRINT, 'SUCCESS' ELSE PRINT, 'Failure'
HDF_VG_DETACH, vg_handle
HDF_CLOSE, file_id
```

IDL Output

```
SUCCESS
```

Version History

4.0	Introduced

See Also

HDF_SD_CREATE, HDF_SD_NAMETOINDEX, HDF_SD_REFTOINDEX, HDF_SD_SELECT, HDF_VG_ADDTR, HDF_VG_ATTACH, HDF_VG_DETACH, HDF_VG_GETID, HDF_VG_INQTR

HDF_SD_ISCOORDVAR

The HDF_SD_ISCOORDVAR function determines whether or not the specified dataset ID represents a NetCDF “coordinate” variable.

Syntax

Result = HDF_SD_ISCOORDVAR(*SDdataset_id*)

Return Value

Returns True (1) if the supplied data set ID is a NetCDF coordinate variable. Otherwise, False (0) is returned.

Arguments

SDdataset_id

An SD dataset ID as returned by HDF_SD _SELECT or HDF_SD_CREATE.

Keywords

None

Version History

4.0	Introduced

HDF_SD_NAMETOINDEX

The HDF_SD_NAMETOINDEX function returns an SD dataset index given its name and SD interface ID. An error message is printed if the dataset cannot be located. The returned index can be used by HDF_SD_SELECT to access an SD dataset.

Syntax

Result = HDF_SD_NAMETOINDEX(*SDinterface_id*, *SDS_Name*)

Return Value

Returns the specified SD dataset index number.

Arguments

SDinterface_id

An SD interface ID as returned by HDF_SD_START.

SDS_Name

A string containing the name of the SD dataset be located.

Keywords

None

Examples

```
; Start the SD interface:
SDinterface_id = HDF_SD_START('demo.hdf')
; Return the index of the 'variable_2' dataset:
index = HDF_SD_NAMETOINDEX(SDinterface_id, 'variable_2')
; Access the dataset:
SDdataset_id=HDF_SD_SELECT(SDinterface_id,index)
; End access:
HDF_SD_ENDACCESS, SDdataset_id
HDF_SD_END, SDinterface_id
```

Version History

4.0	Introduced

See Also

HDF_SD_REFTOINDEX, HDF_SD_SELECT, HDF_SD_START

HDF_SD_REFTOINDEX

The HDF_SD_REFTOINDEX function returns a scientific dataset's index given its reference number and SD interface ID.

Syntax

Result = HDF_SD_REFTOINDEX(*SDinterface_id*, *Reference_number*)

Return Value

Returns the index number associated with the specified SD dataset.

Arguments

SDinterface_id

An SD interface ID as returned by HDF_SD_START.

Reference_number

The SD reference number for the desired dataset.

Keywords

None

Examples

```
; Initialize the SD interface:
SDinterface_id = HDF_SDSTART('demo.hdf')
; Define the reference number for which we want to search:
Reference_number = 66
; Return the index number:
index = HDF_SD_REFTOINDEX(SDinterface_id, Reference_number)
; Now the dataset can be accessed:
SDdataset_id = HDF_SD_SELECT(SDinterface_id, index)
; End access:
HDF_SD_ENDACCESS,SDdataset_id
HDF_SD_END, SDinterface_id
```

Version History

4.0	Introduced

See Also

HDF_SD_IDTOREF, HDF_SD_NAMETOINDEX

HDF_SD_SELECT

The HDF_SD_SELECT function returns an SD dataset ID given the current SD interface ID, and the zero-based SD dataset index.

HDF_SD_FILEINFO can be used to determine the number of SD datasets in an HDF file, HDF_SD_REFTOINDEX can be used to find the index from its SD dataset ID, and HDF_SD_NAMETOINDEX can be used to find the index from its name.

Syntax

Result = HDF_SD_SELECT(*SDinterface_id*, *Number*)

Return Value

Returns the specified SD dataset's identifier.

Arguments

SDinterface_id

A SD interface ID as returned from HDF _SD_START.

Number

A zero-based SD dataset index.

Keywords

None

Examples

```
; Open an HDF file:
SDinterface_id = HDF_SD_START('test.hdf')
; Access the first SD in the HDF file:
SDdataset_id_1=HDF_SD_SELECT(SDinterface_id, 0)
; End access to any SD ids:
HDF_SD_ENDACCESS, SDdataset_id_1
; Close the file:
HDF_SD_END, SDinterface_id
```

Version History

4.0	Introduced

See Also

HDF_SD_CREATE, HDF_SD_END, HDF_SD_ENDACCESS, HDF_SD_NAMETOINDEX, HDF_SD_REFTOINDEX, HDF_SD_SELECT, HDF_SD_START

HDF_SD_SETCOMPRESS

The HDF_SD_SETCOMPRESS procedure compresses an existing HDF SD dataset or sets the compression method of a newly created HDF SD dataset. Available compression methods are No Compression, run-length encoding (RLE), adaptive (skipping) huffman, and GZIP compression. All of these compression methods are lossless. When using skipping huffman compression, IDL automatically determines the correct skipping size. The EFFORT keyword determines the effort applied when using GZIP compression (i.e., when comptype is 4). In general, the default GZIP compression method is the best combination of speed and file size reduction.

Syntax

HDF_SD_SETCOMPRESS, *SDdataset_id*, *comptype* [, EFFORT=*integer*{1 to 9}]

Arguments

SDdataset_id

The HDF SD dataset id as returned by HDF_SD_CREATE or HDF_SD_SELECT.

Comptype

The compression type to be applied to the HDF SD dataset. Allowable values are:

- 0 = NONE (no compression)
- 1 = RLE (run-length encoding)
- 3 = SKIPPING HUFFMAN
- 4 = GZIP

Note

All compression types are lossless.

Keywords

EFFORT

If the comptype is set to 4 (GZIP), then this keyword specifies the effort that GZIP expends in compressing the dataset. The EFFORT keyword is restricted to the range

1 (minimal compression, fastest) to 9 (most compressed, slowest). The default is EFFORT=5.

Examples

```
; Create an HDF SD file:
SDinterface_id = HDF_SD_START('compress.hdf', /CREATE)
; Create an SDS dataset:
SDdataset_id = HDF_SD_CREATE(SDinterface_id, 'dataset1', [9,40],
/LONG)
; Maximal GZIP compression:
HDF_SD_SETCOMPRESS,SDdataset_id,4,EFFORT=9
; Write the data to be compressed:
HDF_SD_ADDDATA,SDdataset_id,fix(dist(9,40))
; End access to the SDS:
HDF_SD_ENDACCESS, SDdataset_id
; End access to the SD interface:
HDF_SD_END, SDinterface_id
```

Note

Compression of HDF SD datasets is a new feature as of HDF 4.1r2 / IDL 5.2.1. Attempts to read HDF SD datasets not created with HDF 4.1r2 (IDL 5.1) or greater will give unpredictable results. Attempts to read HDF compressed SD datasets with IDL versions prior to IDL 5.1, or other HDF readers that use an HDF version prior to HDF 4.1r2, will fail.

Version History

5.2.1	Introduced

HDF_SD_SETEXTFILE

The HDF_SD_SETEXTFILE procedure moves data values from a dataset into an external file. Only the data is moved—all other information remains in the original file. This routine can only be used with HDF version 3.3 (and later) files, not on older HDF files or NetCDF files. Data can only be moved once, and the user must keep track of the external file(s). The OFFSET keyword allows writing to an arbitrary location in the external file.

As shown in the example, when adding data to an external file SD, you *must* first use HDF_SD_ENDACCESS to sync the file, then reacquire the SDS ID with HDF_SD_SELECT before using HDF_SD_SETEXTFILE.

Syntax

HDF_SD_SETEXTFILE, *SDdataset_id*, *Filename* [, OFFSET=*bytes*]

Arguments

SDdataset_id

An SD dataset ID as returned by HDF_SD_SELECT.

Filename

The name of the external file to be written.

Keywords

OFFSET

Set this keyword to a number of bytes from the beginning of the external file at which data writing should begin. Exercise extreme caution when using this keyword with existing files.

Examples

```
; Create an HDF file:
SDinterface_id = HDF_SD_START('ext_main.hdf', /CREATE)
; Add an SD:
SDdataset_id = HDF_SD_CREATE(SDinterface_id, 'float_findgen',
[3,5], /FLOAT)
; Put some data into the SD:
```

```
HDF_SD_ADDDATA, SDdataset_id, FINDGEN(3,5)
; Call HDF_SD_ENDACCESS to sync the file:
HDF_SD_ENDACCESS,SDdataset_id
; Reacquire the SDdataset_id:
SDdataset_id = HDF_SD_SELECT(SDinterface_id, 0)
; Move data to an external file named findgen.hdf:
HDF_SD_SETEXTFILE, SDdataset_id, 'findgen.hdf'
; Retrieve data from the external file into the variable fout:
HDF_SD_GETDATA, SDdataset_id, fout
; Print the contents of fout:
PRINT, fout
; Sync and close the files:
HDF_SD_ENDACCESS, SDdataset_id
HDF_SD_END, SDinterface_id
```

IDL Output

```
0.00000 1.00000 2.00000
3.00000 4.00000 5.00000
6.00000 7.00000 8.00000
9.00000 10.0000 11.0000
12.0000 13.0000 14.0000
```

Version History

4.0	Introduced

See Also

HDF_SD_END, HDF_SD_ENDACCESS, HDF_SD_SELECT, HDF_SD_START

HDF_SD_SETINFO

The HDF_SD_SETINFO procedure sets information about an SD dataset.

Warning

Setting a label, unit, format, or coordinate system string that has more than 256 characters can have unpredictable results.

Syntax

HDF_SD_SETINFO, *SDdataset_id* [, CALDATA=*structure*] [, COORDSYS=*string*] [, FILL=*value*] [, FORMAT=*string*] [, LABEL=*string*] [, RANGE=*[max, min]*] [, UNIT=*string*]

Arguments

SDdataset_id

An SD dataset ID as returned by HDF_SD _SELECT or HDF_SD_CREATE.

Keywords

CALDATA

Set this keyword to a structure that contains the calibration data. This structure must contain five tags as shown below. The first four tags are of double-precision floating-point type. The fifth tag should be a long integer that specifies the HDF number type. The structure should have the following form:

```
CALDATA={ Cal: 0.0D $          ;Calibration Factor
      Cal_Err: 0.0D $          ;Calibration Error
       Offset: 0.0D $          ;Uncalibrated Offset
       Offset_Err: 0.0D $      ;Uncalibrated Offset Error
       Num_Type: 0L }           ;Number Type of Uncalibrated Data
```

The relationship between HDF and IDL number types is illustrated by the following table:

HDF Number Type	IDL Data Type
0L	UNDEFINED
3L	STRING
21L	BYTE
22L	INTEGER
24L	LONG INTEGER
5L	FLOATING-POINT
6L	DOUBLE-PRECISION

Table 4-10: HDF Number Types vs. IDL Data Types

The relationship between the calibrated data (CD) and the uncalibrated data (UD) is given by the equation:

```
CD = Cal * (UD - Offset)
```

Cal and Offset are the values of the Cal and Offset structure fields described above.

COORDSYS

Set this keyword to a string to be used as the SD dataset coordinate system.

FILL

Set this keyword to the fill value of the SD dataset.

FORMAT

Set this keyword to a string to be used as the SD dataset format.

LABEL

Set this keyword to a string to be used as the SD dataset label.

RANGE

Set this keyword to a two dimensional array that contains the minimum and maximum values of the SD dataset.

UNIT

Set this keyword to a string to be used as the SD dataset units.

Examples

```
; Open an HDF file:
SDinterface_id = HDF_SD_START('demo.hdf', /RDWR)
; Define a new dataset for the file:
SDdataset_id = HDF_SD_CREATE(SDinterface_id, 'variable1', [10,
20], /DOUBLE)
; Create a calibration data structure:
CAL={Cal:1.0D, Cal_Err:0.1D, Offset:2.5D, Offset_Err:0.1D, $
   Num_Type:6L}
; Set information about the dataset:
HDF_SD_SETINFO, SDdataset_id, LABEL='label1', unit='unit1', $
   format='format1', coordsys='coord1', FILL=999.991, $
   RANGE=[99.99,-78], caldata=CAL
; Retrieve the information:
HDF_SD_GETINFO, SDdataset_id, LABEL=l, UNIT=u, FORMAT=f, $
   COORDSYS=c, FILL=fill, RANGE=r, CALDATA=cd, $
   NDIMS=ndims, DIMS=dims, TYPE=ty
; Print information about the returned variables:
HELP, l, u, f, c, fill, r, cd, ndims, dims, ty
; Print the range:
PRINT, r
; Print the calibration data:
PRINT, cd
; Print the dimensions:
PRINT, dims
; Close the SD interface:
HDF_SD_ENDACCESS, SDdataset_id
HDF_SD_END, SDinterface_id
```

IDL Output

```
L STRING = 'label1'
U STRING = 'unit1'
F STRING = 'format1'
C STRING = 'coord1'
FILL DOUBLE = 999.99103
R DOUBLE = Array(2)
CD STRUCT = -> < Anonymous > Array(1)
NDIMS LONG = 2
DIMS LONG = Array(2)
TY STRING = 'DOUBLE'

-78.000000 99.989998
```

```
{ 1.0000000 0.10000000 2.5000000 0.10000000 6}

10 20
```

Version History

4.0	Introduced

See Also

HDF_SD_END, HDF_SD_ENDACCESS, HDF_SD_GETINFO, HDF_SD_START

HDF_SD_START

The HDF_SD_START function opens or creates an HDF file and initializes the SD interface.

Note that every file opened with HDF_SD_START should eventually be closed with a call to HDF_SD_END.

Syntax

Result = HDF_SD_START(*Filename* [, /READ | , /RDWR] [, /CREATE])

Return Value

The returned value of this function is the SD ID of the HDF file. If no keywords are present, the file is opened in read-only mode.

Arguments

Filename

A scalar string containing the name of the file to be opened or created. HDF_SD_START can open the following file types: XDR-based NetCDF files, “old-style” DFSD files, or “new-style” SD files. New files are created as “new-style” SD files.

Keywords

READ

Set this keyword to open the SD interface in read-only mode. If no keywords are specified, this is the default behavior.

RDWR

Set this keyword to open the SD interface in read and write mode.

CREATE

Set this keyword to create a new SD file.

Examples

```
; Open a new HDF file. The file is ready to be accessed:
SDinterface_id = HDF_SD_START('test.hdf', /CREATE)
; When finished with the file, close it with a call to HDF_SD_END:
HDF_SD_END, SDinterface_id
```

For a more complicated example, see the documentation for HDF_SD_ATTRSET.

Version History

4.0	Introduced

See Also

HDF_CLOSE, HDF_OPEN, HDF_SD_ATTRFIND, HDF_SD_ATTRINFO, HDF_SD_ATTRSET, HDF_SD_CREATE, HDF_SD_END, HDF_SD_FILEINFO, HDF_SD_NAMETOINDEX, HDF_SD_REFTOINDEX, HDF_SD_SELECT, HDF_SD_SETEXTFILE

HDF_UNPACKDATA

This procedure unpacks an array of byte data into a number of IDL variables. It is useful in deconstructing the output of multi-field HDF Vdata reading routines, such as those found in HDF-EOS, into a set of IDL variables. The packed data is assumed to be an array of bytes that is organized as a number of records. Each record consists of one or more data fields. A record is defined using the HDF_TYPE and HDF_ORDER keywords. These define the record layout in terms of HDF data types. The procedure walks through the input array and copies the values into output IDL arrays. There must be as many entries in the HDF_TYPE and HDF_ORDER keywords as there are data arguments.

Syntax

HDF_UNPACKDATA, *packeddata*, *data1* [, *data2* [, *data3* [, *data4* [, *data5* [, *data6* [, *data7* [, *data8*]]]]]]] [, HDF_ORDER=*array*] [, HDF_TYPE=*array*] [, NREC=*records*]

Arguments

packeddata

A BYTE array of packed data.

data1...data8

These arguments return IDL arrays of the types specified by HDF_TYPE with values for each record in the packed data. If HDF_ORDER is greater than one, the returned array will be 2D and the leading dimension will be of length HDF_ORDER. The one exception is string types, which will be returned as a 1D array of IDL strings. The fixed-length string field is returned as an IDL string up to the first zero value (if present). The trailing dimension will be equal to the minimum of the NREC keyword value or the number of complete records that fit in the packeddata array.

Keywords

HDF_ORDER

Set this keyword to an array with the same length as the number of data fields. The values in the array are equal to the number of elements in the return argument for each record. In the case of strings, this is the length (in characters) of the string to be read. A value of zero is interpreted as one element. The default for this keyword is an array of ones.

HDF_TYPE

Set this keyword to an array with the same length as the number of data fields. The value in the array is an HDF data type for each return argument. The returned IDL variables will have these types. The default for this keyword is an array of the value 5 (an HDF 32-bit float). See “IDL and HDF Data Types” on page 317 for valid values.

NREC

Set this keyword to the number of records to read from packeddata. The default is to read as many complete records as exist in the packeddata array.

Examples

```
a = INDGEN(5)
b = FINDGEN(5)
c = ['This', 'is', 'a', 'string', 'array.']
HELP, a, b, c
hdftype = [ 22, 5, 4] ; HDF INT16, FLOAT32 and CHAR
order = [ 0, 0, 6] ;  2 + 4 + 6 = 12 bytes/record
data = HDF_PACKDATA( a, b, c, HDF_TYPE=hdftype, HDF_ORDER=order)
HELP, data ; a [12, 5] array (5 - 12byte records)
HDF_UNPACKDATA, data, d, e, f, HDF_TYPE=hdftype, HDF_ORDER=order
HELP, d, e, f ; recover the original arrays
```

Version History

5.2	Introduced

See Also

HDF_PACKDATA, HDF_VD_READ, EOS_PT_READLEVEL

HDF_VD_ATTACH

The HDF_VD_ATTACH function accesses a VData with the given Id in an HDF file.

Syntax

Result = HDF_VD_ATTACH(*FileHandle*, *VData_id* [, /READ] [, /WRITE])

Return Value

If successful, a handle for that VData is returned, otherwise 0 is returned.

Arguments

FileHandle

The HDF file handle returned from a previous call to HDF_OPEN.

VData_id

The VData reference number, usually obtained by HDF_VD_GETID or HDF_VD_LONE. Set this argument to -1 to create a new VData.

Keywords

READ

Set this keyword to open the VData for reading. This is the default.

WRITE

Set this keyword to open the VData for writing. If VData_id is set equal to -1, the file is opened for writing whether or not this keyword is set.

Version History

4.0	Introduced

See Also

HDF_VD_DETACH

HDF_VD_ATTRFIND

The HDF_VD_ATTRFIND function returns an attribute's index number given the name of an attribute associated with the specified VData or VData/field pair.

Syntax

Result = HDF_VD_ATTRFIND(*VData*, *FieldID*, *Name*)

Return Value

Returns the specified attribute’s index number or –1 if the attribute cannot be located.

Arguments

VData

The VData handle returned by a previous call to HDF_VD_ATTACH.

FieldID

A zero-based index specifying the field, or a string containing the name of the field within the VData to which the attribute is attached. Setting FieldID to –1 specifies that the attribute is attached to the VData itself.

Name

A string containing the name of the attribute whose index is to be returned.

Keywords

None

Examples

For an example using this routine, see the documentation for HDF_VD_ATTRSET.

Version History

5.5	Introduced

See Also

HDF_VD_ATTRINFO, HDF_VD_ATTRSET, HDF_VD_ISATTR, HDF_VD_NATTRS

HDF_VD_ATTRINFO

The HDF_VD_ATTRINFO procedure reads or retrieves information about a VData attribute or a VData field attribute from the currently attached HDF VData structure. If the attribute is not present, an error message is printed.

Syntax

HDF_VD_ATTRINFO, *VData*, *FieldID*, *AttrID* [, COUNT=*variable*]
[, DATA=*variable*] [, HDF_TYPE=*variable*] [, NAME=*variable*]
[, TYPE=*variable*]

Arguments

VData

The VData handle returned by a previous call to HDF_VD_ATTACH.

FieldID

A zero-based index specifying the field, or a string containing the name of the field within the VData whose attribute is to be read. Setting FieldID to -1 specifies that the attribute to be read is attached to the VData itself.

AttrID

A zero-based integer index specifying the attribute to be read, or a string containing the name of that attribute.

Keywords

COUNT

Set this keyword to a named variable in which the number of data values (order of the attribute) is returned.

DATA

Set this keyword to a named variable in which the attribute data is returned.

HDF_TYPE

Set this keyword to a named variable in which the HDF data type of the attribute is returned as a scalar string. See "IDL and HDF Data Types" on page 317 for valid values.

NAME

Set this keyword to a named variable in which the name of the attribute is returned.

TYPE

Set this keyword to a named variable in which the IDL type of the attribute is returned as a scalar string.

Examples

For an example using this routine, see the documentation for HDF_VD_ATTRSET.

Version History

5.5	Introduced

See Also

HDF_VD_ATTRFIND, HDF_VD_ATTRSET, HDF_VD_ISATTR, HDF_VD_NATTRS

HDF_VD_ATTRSET

The HDF_VD_ATTRSET procedure writes a VData attribute or a VData field attribute to the currently attached HDF VData structure. If no data type keyword is specified, the data type of the attribute value is used.

Syntax

HDF_VD_ATTRSET, *VData*, *FieldID*, *Attr_Name*, *Values* [, *Count*] [, /BYTE] [, /DFNT_CHAR8] [, /DFNT_FLOAT32] [, /DFNT_FLOAT64] [, /DFNT_INT8] [, /DFNT_INT16] [, /DFNT_INT32] [, /DFNT_UCHAR8] [, /DFNT_UINT8] [, /DFNT_UINT16] [, /DFNT_UINT32] [, /DOUBLE] [, /FLOAT] [, /INT] [, /LONG] [, /SHORT] [, /STRING] [, /UINT] [, /ULONG]

Arguments

VData

The VData handle returned by a previous call to HDF_VD_ATTACH.

Note

The VData structure must have been attached in write mode in order for attributes to be correctly associated with a VData or one of its fields. If the VData is not write accessible, HDF does not return an error; instead, the attribute information is written to the file but is not associated with the VData.

FieldID

A zero-based index specifying the field, or a string containing the name of the field within the VData whose attribute is to be set. If FieldID is set to -1, the attribute will be attached to the VData itself.

Attr_Name

A string containing the name of the attribute to be written.

Values

The attribute value(s) to be written.

Note

Attributes to be written as characters may not be a multi-dimensional array (e.g. if being converted from byte values) or an array of IDL strings.

Count

An optional integer argument specifying how many values are to be written. Count must be less than or equal to the number of elements in the Values argument. If not specified, the actual number of values present will be written.

Keywords

BYTE

Set this keyword to indicate that the attribute is composed of bytes. Data will be stored with the HDF DFNT_UINT8 data type. Setting this keyword is the same as setting the DFNT_UINT8 keyword.

DFNT_CHAR8

Set this keyword to create an attribute of HDF type DFNT_CHAR8. Setting this keyword is the same as setting the STRING keyword.

DFNT_FLOAT32

Set this keyword to create an attribute of HDF type DFNT_FLOAT32. Setting this keyword is the same as setting the FLOAT keyword.

DFNT_FLOAT64

Set this keyword to create an attribute of HDF type DFNT_FLOAT64. Setting this keyword is the same as setting the DOUBLE keyword.

DFNT_INT8

Set this keyword to create an attribute of HDF type DFNT_INT8.

DFNT_INT16

Set this keyword to create an attribute of HDF type DFNT_INT16. Setting this keyword is the same as setting either the INT keyword or the SHORT keyword.

DFNT_INT32

Set this keyword to create an attribute of HDF type DFNT_INT32. Setting this keyword is the same as setting the LONG keyword.

DFNT_UCHAR8

Set this keyword to create an attribute of HDF type DFNT_UCHAR8.

DFNT_UINT8

Set this keyword to create an attribute of HDF type DFNT_UINT8. Setting this keyword is the same as setting the BYTE keyword.

DFNT_UINT16

Set this keyword to create an attribute of HDF type DFNT_UINT16.

DFNT_UINT32

Set this keyword to create an attribute of HDF type DFNT_UINT32.

DOUBLE

Set this keyword to indicate that the attribute is composed of double-precision floating-point values. Data will be stored with the HDF type DFNT_FLOAT64. Setting this keyword is the same as setting the DFNT_FLOAT64 keyword.

FLOAT

Set this keyword to indicate that the attribute is composed of single-precision floating-point values. Data will be stored with the HDF type DFNT_FLOAT32 data type. Setting this keyword is the same as setting the DFNT_FLOAT32 keyword.

INT

Set this keyword to indicate that the attribute is composed of 16-bit integers. Data will be stored with the HDF type DFNT_INT16 data type. Setting this keyword is the same as setting either the SHORT keyword or the DFNT_INT16 keyword.

LONG

Set this keyword to indicate that the attribute is composed of longword integers. Data will be stored with the HDF type DFNT_INT32 data type. Setting this keyword is the same as setting the DFNT_INT32 keyword.

SHORT

Set this keyword to indicate that the attribute is composed of 16-bit integers. Data will be stored with the HDF type DFNT_INT16 data type. Setting this keyword is the same as setting either the INT keyword or the DFNT_INT16 keyword.

STRING

Set this keyword to indicate that the attribute is composed of strings. Data will be stored with the HDF type DFNT_CHAR8 data type. Setting this keyword is the same as setting the DFNT_CHAR8 keyword.

UINT

Set this keyword to indicate that the attribute is composed of unsigned 2-byte integers. Data will be stored with the HDF type DFNT_UINT16 data type. Setting this keyword is the same as setting the DFNT_UINT16 keyword.

ULONG

Set this keyword to indicate that the attribute is composed of unsigned longword integers. Data will be stored with the HDF type DFNT_UINT32 data type. Setting this keyword is the same as setting the DFNT_UINT32 keyword.

Examples

```
; Open an HDF file.
fid = HDF_OPEN(FILEPATH('vattr_example.hdf',$
   SUBDIRECTORY = ['examples', 'data']), /RDWR)

; Locate and attach an existing vdata.
vdref = HDF_VD_FIND(fid, 'MetObs')
vdid = HDF_VD_ATTACH(fid, vdref, /WRITE)

; Attach two attributes to the vdata.
HDF_VD_ATTRSET, vdid, -1, 'vdata_contents', $
   'Ground station meteorological observations.'
HDF_VD_ATTRSET, vdid, -1, 'num_stations', 10

; Attach an attribute to one of the fields in the vdata.
HDF_VD_ATTRSET, vdid, 'TempDP', 'field_contents', $
   'Dew point temperature in degrees Celsius.'

; Get the number of attributes associated with the vdata.
num_vdattr = HDF_VD_NATTRS(vdid, -1)
```

```
PRINT, 'Number of attributes attached to vdata MetObs: ', $
   num_vdattr

; Get information for one of the vdata attributes by first finding
; the attribute's index number.
attr_index = HDF_VD_ATTRFIND(vdid, -1, 'vdata_contents')
HDF_VD_ATTRINFO, vdid, 1, attr_index, $
   NAME = attr_name,DATA = metobs_contents
HELP, attr_name, metobs_contents

; Get information for another vdata attribute using the
; attribute's name.
HDF_VD_ATTRINFO, vdid, -1, 'num_stations', DATA = num_stations, $
   HDF_TYPE = hdftype, TYPE = idltype
HELP, num_stations, hdftype,idltype
PRINT, num_stations

; Get the number of attributes attached to the vdata field
; TempDP.
num_fdattr = HDF_VD_NATTRS(vdid, 'TempDP')
PRINT, 'Number of attributes attached to field TempDP: ', $
   num_fdattr

; Get the information for the vdata field attribute.
HDF_VD_ATTRINFO, vdid, 'TempDP', 'field_contents', $
   COUNT = count, HDF_TYPE = hdftype, TYPE = idltype, $
   DATA = dptemp_attr
HELP, count, hdftype, idltype, dptemp_attr

; End access to the vdata.
HDF_VD_DETACH, vdid

; Attach a vdata which stores one of the attribute values.
vdid = HDF_VD_ATTACH(fid, 5)

; Get the vdata's name and check to see that it is indeed storing
; an attribute.
HDF_VD_GET, vdid, NAME = vdname
isattr = HDF_VD_ISATTR(vdid)
HELP, vdname, isattr

; End access to the vdata and the HDF file.
HDF_VD_DETACH, vdid
HDF_CLOSE, fid
```

IDL Output

```
Number of attributes attached to vdata MetObs:            2
ATTR_NAME       STRING    = 'vdata_contents'
```

```
METOBS_CONTENTS STRING    = 'Ground station meteorological
observations.'
NUM_STATIONS    INT       = Array[1]
HDFTYPE         STRING    = 'DFNT_INT16'
IDLTYPE         STRING    = 'INT'
      10
Number of attributes attached to field TempDP:            1
COUNT           LONG      =           41
HDFTYPE         STRING    = 'DFNT_CHAR8'
IDLTYPE         STRING    = 'STRING'
DPTEMP_ATTR     STRING    = 'Dew point temperature in degrees
Celsius.'
VDNAME          STRING    = 'field_contents'
ISATTR          LONG      =            1
```

Version History

5.5	Introduced

See Also

HDF_VD_ATTRFIND, HDF_VD_ATTRINFO, HDF_VD_ISATTR, HDF_VD_NATTRS

HDF_VD_DETACH

The HDF_VD_DETACH procedure is called when done accessing a VData in an HDF file. This routine must be called for every VData attached for writing before closing the HDF file to insure that VSET information is properly updated.

Syntax

HDF_VD_DETACH, *VData*

Arguments

VData

The VData handle returned by a previous call to HDF_VD_ATTACH.

Keywords

None

Version History

4.0	Introduced

See Also

HDF_VD_ATTACH

HDF_VD_FDEFINE

The HDF_VD_FDEFINE procedure adds a new field specification for a VData in an HDF file. HDF_VD_FDEFINE can only be used for a new VData.

Syntax

HDF_VD_FDEFINE, *VData*, *Fieldname* [, /BYTE | , /DOUBLE | , /FLOAT | , /INT | , /LONG] [, ORDER=*value*] [, /UINT] [, /ULONG]

Arguments

VData

The VData handle returned by a previous call to HDF_VD_ATTACH.

Fieldname

A string containing the new field name.

Keywords

BYTE

Set this keyword to indicate that the field will contain 8-bit unsigned integer data.

DOUBLE

Set this keyword to indicate that the field will contain 64-bit floating point data.

FLOAT

Set this keyword to indicate that the field will contain 32-bit floating point data.

INT

Set this keyword to indicate that the field will contain 16-bit integer data.

LONG

Set this keyword to indicate that the field will contain 32-bit integer data.

ORDER

This keyword specifies the number of distinct components in the new field. Compound variables have an order greater than 1. The default order is 1.

UINT

Set this keyword to indicate that the field will contain 16-bit unsigned integer data.

ULONG

Set this keyword to indicate that the field will contain 32-bit unsigned integer data.

Examples

```
HDF_VD_FDEFINE, vid, 'VEL', /DOUBLE, ORDER=3
```

Version History

4.0	Introduced

HDF_VD_FEXIST

The HDF_VD_FEXIST function determines whether the specified fields exist in the given HDF file.

Syntax

Result = HDF_VD_FEXIST(*VData*, *Fieldnames*)

Return Value

Returns 1 (True) if all the specified fields exist.

Arguments

VData

The VData handle returned by a previous call to HDF_VD_ATTACH.

Fieldnames

A string containing a comma-separated list of fields to test. For example, 'VEL' or 'PZ,PY,PX'.

Keywords

None

Version History

4.0	Introduced

HDF_VD_FIND

The HDF_VD_FIND function returns the reference number of a VData with the specified name in an HDF file.

Syntax

Result = HDF_VD_FIND(*FileHandle*, *Name*)

Return Value

Returns the reference number of the named VData. A 0 is returned if an error occurs or a VData of the given name does not exist.

Arguments

FileHandle

The HDF file handle returned from a previous call to HDF_OPEN.

Name

A string containing the name of the VData to be found.

Keywords

None

Version History

4.0	Introduced

HDF_VD_GET

The HDF_VD_GET procedure is a general VData inquiry routine. Set the various keywords to named variables to return information about a VData in an HDF file.

Syntax

HDF_VD_GET, *VData* [, CLASS=*variable*] [, COUNT=*variable*]
[, FIELDS=*variable*] [, INTERLACE=*variable*] [, NAME=*variable*]
[, NFIELDS=*variable*] [, REF=*variable*] [, SIZE=*variable*] [, TAG=*variable*]

Arguments

VData

A VData handle returned by HDF_VD_ATTACH.

Keywords

CLASS

Set this keyword to a named variable in which the class name of the VData is returned as a string.

COUNT

Set this keyword to a named variable in which a long, containing the number of records in the VData, is returned.

FIELDS

Set this keyword to a named variable in which a comma-separated string of fields in the VData is returned (e.g., 'PX,PY,PZ')

The maximum number of fields is 256. Each field can be up to 128 characters in length. The returned fields may or may not contain buffering whitespace depending on how the HDF file was created.

INTERLACE

Set this keyword to a named variable in which a string, containing either 'FULL_INTERLACE' or 'NO_INTERLACE', is returned.

NAME

Set this keyword to a named variable in which a string, containing the name of the VData, is returned.

NFIELDS

Set this keyword to a named variable in which a long, containing the number of fields in the VDATA, is returned. For example, the VData containing the fields “PX,PY,PZ”, has an NFIELDS of 3.

REF

Set this keyword to a named variable in which the reference number of the VData is returned.

SIZE

Set this keyword to a named variable in which a long, containing the local size of a record of VData, is returned.

TAG

Set this keyword to a named variable in which the tag number of the VData is returned.

Examples

```
HDF_VD_GET, vdat, CLASS=c, COUNT=co, FIELDS=f, NAME=n, SIZE=s
```

Version History

4.0	Introduced

See Also

HDF_VD_GETINFO, HDF_VG_GETINFO

HDF_VD_GETID

The HDF_VD_GETID function returns the VData reference number for the next VData in an HDF file after the specified *VData_id*.

Set VData_id to -1 to return the first VData ID in the file.

Syntax

Result = HDF_VD_GETID(*FileHandle*, *VData_id*)

Return Value

Returns the next VData reference number.

Arguments

FileHandle

The HDF file handle returned by a previous call to HDF_OPEN.

VData_id

The VData reference number, generally obtained by HDF_VD_GETID or HDF_VD_LONE. Set this argument to -1 to return the first VData in the file.

Keywords

None

Version History

4.0	Introduced

HDF_VD_GETINFO

The HDF_VD_GETINFO procedure is a general VData inquiry routine. Set the various keywords to named variables to return information about each field of a VData in a HDF file.

Syntax

HDF_VD_GETINFO, *VData*, *Index* [, NAME=*variable*] [, ORDER=*variable*] [, SIZE=*variable*] [, TYPE=*variable*]

Arguments

VData

The VData handle returned by a previous call to HDF_VD_ATTACH.

Index

A zero-based index specifying which field or the name of the field within the VData to inquire about. For example:

```
HDF_VD_GETINFO, Vdat, 'VEL', ORDER=order
```

Keywords

NAME

Set this keyword to a named variable in which the name of the field is returned as a string.

ORDER

Set this keyword to a named variable in which the order of the field is returned.

SIZE

Set this keyword to a named variable in which the size of a data value for the specified field in the VData is returned.

TYPE

Set this keyword to a named variable in which the type of the field is returned. One of the following strings is returned: 'BYTE', 'INT', 'LONG', 'FLOAT', 'DOUBLE'.

Examples

```
HDF_VD_GET, Vdat, NFIELDS=n
FOR index=0,n-1 DO BEGIN
  HDF_VD_GETINFO, Vdat, index, NAME=n, TYPE=t, ORDER=o
  PRINT, index, ':', n, 'TYPE=', t, 'ORDER=', o
ENDFOR
```

Version History

4.0	Introduced

HDF_VD_INSERT

The HDF_VD_INSERT procedure adds a VData or VGroup to the contents of a VGroup in an HDF file.

Syntax

HDF_VD_INSERT, *VGroup*, *VData* [, POSITION=*variable*]

Arguments

VGroup

The VGroup handle returned by a previous call to HDF_VG_ATTACH.

VData

The VData (or VGroup) handle returned by HDF_VD_ATTACH (HDF_VG_ATTACH).

Keywords

POSITION

Set this keyword to return the entry position of the element (VData or VGroup, respectively) within the existing VGroup to which you are adding.

Version History

4.0	Introduced

HDF_VD_ISATTR

The HDF_VD_ISATTR function determines whether or not a VData is storing an attribute. HDF stores attributes as VDatas, so this routine provides a means to test whether or not a particular VData contains an attribute.

Syntax

Result = HDF_VD_ISATTR(*VData*)

Return Value

Returns TRUE (1) if the VData is storing an attribute, FALSE (0) otherwise.

Arguments

VData

The VData handle returned by a previous call to HDF_VD_ATTACH.

Keywords

None

Examples

For an example using this routine, see the documentation for HDF_VD_ATTRSET.

Version History

4.0	Introduced

See Also

HDF_VD_ATTRFIND, HDF_VD_ATTRINFO, HDF_VD_ATTRSET, HDF_VD_NATTRS

HDF_VD_ISVD

The HDF_VD_ISVD function determines if the object associated with *Id* is a VData in an HDF file.

Syntax

Result = HDF_VD_ISVD(*VGroup*, *Id*)

Return Value

Returns True (1) if the object is VData, or False (0) otherwise.

Arguments

VGroup

The VGroup handle returned by a previous call to HDF_VG_ATTACH.

Id

The VGroup reference number obtained by a previous call to HDF_VG_GETNEXT.

Keywords

None

Examples

```
Vid = HDF_VG_GETNEXT(Vgrp, -1)
PRINT, HDF_VD_ISVD(VGrp, Vid)
```

Version History

4.0	Introduced

HDF_VD_ISVG

The HDF_VD_ISVG function determines if the object associated with *Id* is a VGroup in an HDF file.

Syntax

Result = HDF_VD_ISVG(*VGroup*, *Id*)

Return Value

Returns True (1) if the object is a VGroup, or False (0) otherwise.

Arguments

VGroup

The VGroup handle returned by a previous call to HDF_VD_ATTACH.

Id

The VGroup reference number obtained by a previous call to HDF_VG_GETNEXT.

Keywords

None

Examples

```
Vid = HDF_VG_GETNEXT(Vgrp, -1)
PRINT, HDF_VD_ISVG(VGrp, Vid)
```

Version History

4.0	Introduced

HDF_VD_LONE

The HDF_VD_LONE function returns an array containing all VDatas in an HDF file that are not contained in another VData.

Syntax

Result = HDF_VD_LONE(*FileHandle* [, MAXSIZE=*value*])

Return Value

Returns any lone VDatas within an array. If there are no lone VDatas, HDF_VD_LONE returns -1.

Arguments

FileHandle

The HDF file handle returned from a previous call to HDF_OPEN.

Keywords

MAXSIZE

The maximum number of groups to be returned (the default is to return all known lone VDatas). For example, to return only the first 12 groups:

```
X = HDF_VD_LONE(fid, MAX=12)
```

Examples

```
X = HDF_VD_LONE(fid)
IF N_ELEMENTS(X) EQ 0 THEN $
PRINT, 'No Lone VDatas' ELSE PRINT, 'Lone VDatas:', X
```

Version History

4.0	Introduced

HDF_VD_NATTRS

The HDF_VD_NATTRS function returns the number of attributes associated with the specified VData or VData/field pair.

Syntax

Result = HDF_VD_NATTRS(*VData*, *FieldID*)

Return Value

Returns the number of attributes if successful. Otherwise, –1 is returned.

Arguments

VData

The VData handle returned by a previous call to HDF_VD_ATTACH.

FieldID

A zero-based index specifying the field, or a string containing the name of the field, within the VData whose attributes are to be counted. Setting Index to –1 specifies that attributes attached to the VData itself are to be counted.

Keywords

None

Examples

For an example using this routine, see the documentation for HDF_VD_ATTRSET.

Version History

5.5	Introduced

See Also

HDF_VD_ATTRFIND, HDF_VD_ATTRINFO, HDF_VD_ATTRSET, HDF_VD_ISATTR

HDF_VD_READ

The HDF_VD_READ function reads data from a VData in an HDF file.

The default is to use FULL_INTERLACE and to read all fields in all records. The user can override the defaults with keywords. If multiple fields with different data types are read, all of the data is read into a byte array. The data must then be explicitly converted back into the correct type(s) using various IDL type conversion routines. For example:

```
nread = HDF_VD_READ(vdat, x, NREC=1, FIELDS="FLT,LNG")
floatvalue = FLOAT(x, 0)
longvalue = LONG(x, 4)
```

Syntax

Result = HDF_VD_READ(*VData*, *Data* [, FIELDS=*string*] [, /FULL_INTERLACE | , /NO_INTERLACE] [, NRECORDS=*records*])

Return Value

This function returns the number of records successfully read from the VData.

Arguments

VData

The VData handle returned by a previous call to HDF_VD_ATTACH.

Data

A named variable in which the data is returned.

Keywords

FIELDS

A string containing a comma-separated list of fields to be read. Normally HDF_VD_READ will read all fields in the VData.

FULL_INTERLACE

Set this keyword to use full interlace when reading (the default).

NO_INTERLACE

Set this keyword to use no interlace when reading.

NRECORDS

The number of records to read. By default, HDF_VD_READ reads all records from a VData.

Examples

Typical read:

```
NREC = HDF_VD_READ(Vdat, X)
```

Read one field:

```
NREC = HDF_VD_READ(Vdat, X, FIELDS='VEL')
```

Read a record:

```
NREC = HDF_VD_READ(Vdat, X, NRECORDS=1)
```

Version History

4.0	Introduced

HDF_VD_SEEK

The HDF_VD_SEEK procedure moves the read pointer within the specified VData in an HDF file to a specific record number. Note that the record number is zero-based.

Syntax

HDF_VD_SEEK, *VData*, *Record*

Arguments

VData

A VData handle returned by HDF_VD_ATTACH.

Record

The zero-based record number to seek.

Keywords

None

Version History

4.0	Introduced

HDF_VD_SETINFO

The HDF_VD_SETINFO procedure specifies general information about a VData in an HDF file. Keywords can be used to establish the name, class, and interlace for the specified VData.

Syntax

HDF_VD_SETINFO, *VData* [, CLASS=*string*] [, /FULL_INTERLACE | , /NO_INTERLACE] [, NAME=*string*]

Arguments

VData

The VData handle returned by a previous call to HDF_VD_ATTACH.

Keywords

CLASS

A string that sets the class name for the VData.

FULL_INTERLACE

Set this keyword to store data in the file with full interlace (i.e., sequentially by record).

NAME

A string that sets the name of the VData.

NO_INTERLACE

Set this keyword to store data in the file with no interlace (i.e., sequentially by field).

Examples

```
HDF_VD_SETINFO, Vdat, NAME='My Favorite Data', /FULL
```

Version History

4.0	Introduced

HDF_VD_WRITE

The HDF_VD_WRITE procedure stores data in a VData in an HDF file.

There are many restrictions on writing data to a VData. When writing multiple fields of varying types, only limited error checking is possible. When writing a series of fields all with the same type, data is converted to that type before writing. For example:

```
Vdat = HDF_VD_ATTACH(Fid, -1, /WRITE)
; Create a 10 integer vector:
Data = INDGEN(10)
; Data converted to FLOAT before write:
HDF_VD_WRITE, Vdat, 'PX', Data
```

It is possible to write less data than exists in the Data argument by using the NRECORDS keyword. For example, the following command writes 5 records, instead of the 10 implied by the size of the data (VEL is assumed to be of type FLOAT, order=3):

```
HDF_VD_WRITE, Vdat, 'VEL', FINDGEN(3,10),NREC=5
```

VEL now contains [[0.0, 1.0, 2.0], ..., [12.0, 13.0, 14.0]]

HDF_VD_WRITE will not allow a user to specify more records than exist. For example, the following command fails:

```
HDF_VD_WRITE, Vdat, 'VEL', [1,2,3], NREC=1000
```

Known Issues

HDF vdatas can only be appended or overwritten if they are defined at creation with a file interlacing mode of FULL_INTERLACE. Records in a fully interlaced vdata are written record-by-record which allows them to be appended or overwritten. For further information, consult the “Writing to Multi-Field Vdatas” section in the *HDF User's Guide* published by the National Center for Supercomputing (available at http://hdf.ncsa.uiuc.edu/doc.html).

Restrictions

It is not possible to write IDL structures directly to a VData (because of possible internal padding depending upon fields/machine architecture, etc.). The user must put the data into a byte array before using HDF_VD_WRITE.

When writing a series of fields all with the same type, the low order dimension of *Data* must match the sum of the orders of the fields. For example:

```
HDF_VD_WRITE, Vdat, 'PX,PY', FLTARR(3,10)
```

fails. PX and PY are both order 1 (total 2) and the array's low order dimension is 3.

Syntax

HDF_VD_WRITE, *VData*, *Fields*, *Data*
[, /FULL_INTERLACE | , /NO_INTERLACE] [, NRECORDS=*records*]

Arguments

VData

The VData handle returned by a previous call to HDF_VD_ATTACH.

Fields

A string containing a comma-separated list of the fields to be written.

Data

The data to be written to the specified VData.

Keywords

FULL_INTERLACE

Set this keyword to use full interlace when writing (the default).

NO_INTERLACE

Set this keyword to use no interlace when writing.

NRECORDS

The number of records to written. By default, HDF_VD_WRITE writes all records from a VData.

Version History

4.0	Introduced

HDF_VG_ADDTR

The HDF_VG_ADDTR procedure adds a tag and reference to the specified VGroup in an HDF file.

Syntax

HDF_VG_ADDTR, *VGroup*, *Tag*, *Ref*

Arguments

VGroup

The VGroup handle returned by a previous call to HDF_VG_ATTACH.

Tag

The tag to be written.

Reference

The reference number to be written.

Keywords

None

Examples

See "HDF_SD_IDTOREF" on page 498 for an example using this function.

Version History

4.0	Introduced

See Also

HDF_VG_GETTR, HDF_VG_GETTRS, HDF_VG_INQTR, HDF_VG_INSERT

HDF_VG_ATTACH

The HDF_VG_ATTACH function attaches (opens) a VGroup in an HDF file for reading or writing.

If VGroup_id is set to -1, a new VGroup is created. If neither the READ nor WRITE keywords are set, the VGroup is opened for reading.

Syntax

Result = HDF_VG_ATTACH(*FileHandle*, *VGroup_id* [, /READ] [, /WRITE])

Return Value

If successful, a handle for the specified group is returned. If it fails, 0 is returned.

Arguments

FileHandle

The HDF file handle returned from a previous call to HDF_OPEN.

VGroup_id

The VGroup reference number, generally obtained by HDF_VG_GETID or HDF_VG_LONE.

Keywords

READ

Set this keyword to open the VGroup for reading.

WRITE

Set this keyword to open the VGroup for writing.

Examples

See "HDF_SD_IDTOREF" on page 498 for an example using this function.

Version History

4.0	Introduced

See Also

HDF_VG_DETACH

HDF_VG_DETACH

The HDF_VG_DETACH procedure should be called when you are finished accessing a VGroup in an HDF file. This routine must be called for every VGroup attached for writing before closing the HDF file in order to insure that VSET information is properly updated.

Syntax

HDF_VG_DETACH, *VGroup*

Arguments

VGroup

The VGroup handle returned by a previous call to HDF_VG_ATTACH.

Keywords

None

Examples

See "HDF_SD_IDTOREF" on page 498 for an example using this function.

Version History

4.0	Introduced

See Also

HDF_VG_ATTACH

HDF_VG_GETID

The HDF_VG_GETID function returns the VGroup ID for the next VGroup after the specified VGroup_id in an HDF file. Use a VGroup_id of -1 to get the first VGroup in the file.

Syntax

Result = HDF_VG_GETID(*FileHandle*, *VGroup_id*)

Return Value

Returns the next VGroup ID.

Arguments

FileHandle

The HDF file handle returned from a previous call to HDF_OPEN.

VGroup_id

The VGroup reference number, generally obtained by HDF_VG_GETID or HDF_VG_LONE.

Keywords

None

Examples

See "HDF_SD_IDTOREF" on page 498 for an example using this function.

Version History

4.0	Introduced

HDF_VG_GETINFO

The HDF_VG_GETINFO procedure is a general VGroup inquiry routine. Set the various keywords to named variables to return information about different aspects of a VGroup in an HDF file.

Syntax

HDF_VG_GETINFO, *VGroup* [, CLASS=*variable*] [, NAME=*variable*]
[, NENTRIES=*variable*] [, REF=*variable*] [, TAG=*variable*]

Arguments

VGroup

The VGroup handle returned by a previous call to HDF_VG_ATTACH.

Keywords

CLASS

Set this keyword to a named variable in which the class of the VGroup is returned as a string.

NAME

Set this keyword to a named variable in which the name of the VGroup is returned as a string.

NENTRIES

Set this keyword to a named variable in which the number of objects inside the VGroup is returned as a long integer.

REF

Set this keyword to a named variable in which the reference number of the specified Vgroup is returned.

TAG

Set this keyword to a named variable in which the tag number of the specified Vgroup is returned.

Examples

```
HDF_VG_GETINFO, Vgrp, CLASS=c, NAME=nm, NENTRIES=n
PRINT, c, nm, n
```

Version History

4.0	Introduced

HDF_VG_GETNEXT

The HDF_VG_GETNEXT function returns the reference number of the next object inside a VGroup in an HDF file. If Id is -1, the first item in the VGroup is returned, otherwise Id should be set to a reference number previously returned by HDF_VG_GETNEXT.

Syntax

Result = HDF_VG_GETNEXT(*VGroup, Id*)

Return Value

Returns the reference number of the next object or returns -1 if there was an error or there are no more objects after the one specified by Id.

Arguments

VGroup

The VGroup handle returned by a previous call to HDF_VG_ATTACH.

Id

A VGroup or VData reference number obtained by a previous call to HDF_VG_GETNEXT. Alternatively, this value can be set to -1 to return the first item in the VGroup.

Keywords

None

Version History

4.0	Introduced

HDF_VG_GETTR

The HDF_VG_GETTR procedure returns the tag/reference pair at the specified position within a VGroup in an HDF file.

Syntax

HDF_VG_GETTR, *VGroup*, *Index*, *Tags*, *Refs*

Arguments

VGroup

The VGroup handle returned by a previous call to HDF_VG_ATTACH.

Index

The position within *VGroup*.

Tags

A named variable in which the tag numbers are returned.

Refs

A named variable in which the reference numbers are returned.

Keywords

None

Version History

4.0	Introduced

HDF_VG_GETTRS

The HDF_VG_GETTRS procedure returns the tag/reference pairs of the HDF file objects belonging to the specified VGroup.

Syntax

HDF_VG_GETTRS, *VGroup*, *Tags*, *Refs* [, MAXSIZE=*value*]

Arguments

VGroup

The VGroup handle returned by a previous call to HDF_VG_ATTACH.

Tags

A named variable in which the tag numbers are returned.

Refs

A named variable in which the reference numbers are returned.

Keywords

MAXSIZE

The maximum number of tags and references to be returned. The default is to return all tags and references in VGroup.

Version History

4.0	Introduced

HDF_VG_INQTR

The HDF_VG_INQTR function returns true if the specified tag and reference pair is linked to the specified VGroup in an HDF file.

Syntax

Result = HDF_VG_INQTR(*VGroup*, *Tag*, *Ref*)

Return Value

Returns true if the link exists.

Arguments

VGroup

The VGroup handle returned by a previous call to HDF_VG_ATTACH.

Tag

The tag number.

Ref

The reference number.

Keywords

None

Examples

See "HDF_SD_IDTOREF" on page 498 for an example using this function.

Version History

4.0	Introduced

HDF_VG_INSERT

The HDF_VG_INSERT procedure adds a VData or VGroup to the contents of a VGroup in an HDF file.

Syntax

HDF_VG_INSERT, *VGroup*, *VData* [, POSITION=*variable*]

Arguments

VGroup

The VGroup handle returned by a previous call to HDF_VG_ATTACH.

VData

The VData (or VGroup) handle returned by HDF_VD_ATTACH (HDF_VG_ATTACH).

Keywords

POSITION

Set this keyword to return the current position of the element (VData or VGroup, respectively) within the existing data group.

Version History

4.0	Introduced

HDF_VG_ISVD

The HDF_VG_ISVD function returns true if the object associated with Id is a VData in an HDF file.

Syntax

Result = HDF_VG_ISVD(*VGroup*, *Id*)

Return Value

Returns true if the specified object is a VData.

Arguments

VGroup

The VGroup handle returned by a previous call to HDF_VG_ATTACH.

Id

The VGroup or VData reference number obtained by a previous call to HDF_VG_GETNEXT.

Keywords

None

Examples

```
Vid = HDF_VG_GETNEXT(Vgrp, -1)
PRINT, HDF_VG_ISVD(VGrp, Vid)
```

Version History

4.0	Introduced

HDF_VG_ISVG

The HDF_VG_ISVG function returns true if the object associated the Id is a VGroup in an HDF file.

Syntax

Result = HDF_VG_ISVG(*VGroup*, *Id*)

Return Value

Returns true if the specified object is a VGroup.

Arguments

VGroup

The VGroup handle returned by a previous call to HDF_VG_ATTACH.

Id

The VGroup or VData reference number obtained by a previous call to HDF_VG_GETNEXT.

Keywords

Examples

```
Vid = HDF_VG_GETNEXT(Vgrp, -1)
PRINT, HDF_VG_ISVG(VGrp, Vid)
```

Version History

4.0	Introduced

HDF_VG_LONE

The HDF_VG_LONE function returns an array containing the IDs of all VGroups in an HDF file that are not contained in another VGroup.

Syntax

Result = HDF_VG_LONE(*FileHandle* [, MAXSIZE=*value*])

Return Value

Returns the IDs of lone VGroups or returns -1 if there are no lone VGroups,.

Arguments

FileHandle

The HDF file handle returned by a previous call to HDF_OPEN.

Keywords

MAXSIZE

The maximum number of groups to return (the default is to return all lone VGroups). For example, to return no more than 12 VGroups, use the command:

```
X = HDF_VG_LONE(fid, MAX=12)
```

Examples

```
X=HDF_VG_LONE(fid)
IF X(0) EQ-1 THEN $
   PRINT, "No Lone VGroups" ELSE PRINT, "Lone VGroups:", X
```

Version History

4.0	Introduced

HDF_VG_NUMBER

The HDF_VG_NUMBER function returns the number of HDF file objects in the specified VGroup.

Syntax

Result = HDF_VG_NUMBER(*VGroup*)

Return Value

Returns the number of objects.

Arguments

VGroup

The VGroup handle returned by a previous call to HDF_VG_ATTACH.

Keywords

None

Version History

4.0	Introduced

HDF_VG_SETINFO

The HDF_VG_SETINFO procedure sets the name and class of a VGroup.

Syntax

HDF_VG_SETINFO, *VGroup* [, CLASS=*string*] [, NAME=*string*]

Arguments

VGroup

The VGroup handle as returned by HDF_VG _ATTACH.

Keywords

CLASS

A string containing the class name for the VGroup.

NAME

A string containing the name for the VGroup.

Examples

```
fid = HDF_OPEN('demo.hdf',/RDWR) ; Open an HDF file:
vgid = HDF_VG_ATTACH(fid, -1, /WRITE) ; Add a new VGroup:
; Set the name and class for the VGroup:
HDF_VG_SETINFO, vgid, NAME='My Name', CLASS='My VGroup Class'
; Retrieve the name and class information from the file:
HDF_VG_GETINFO, vgid, NAME=outname, CLASS=outclass
; Print information about the returned variables:
HELP, outname, outclass
; End VGroup access:
HDF_VG_DETACH, vgid
; Close the HDF file:
HDF_CLOSE, fid
```

IDL Output

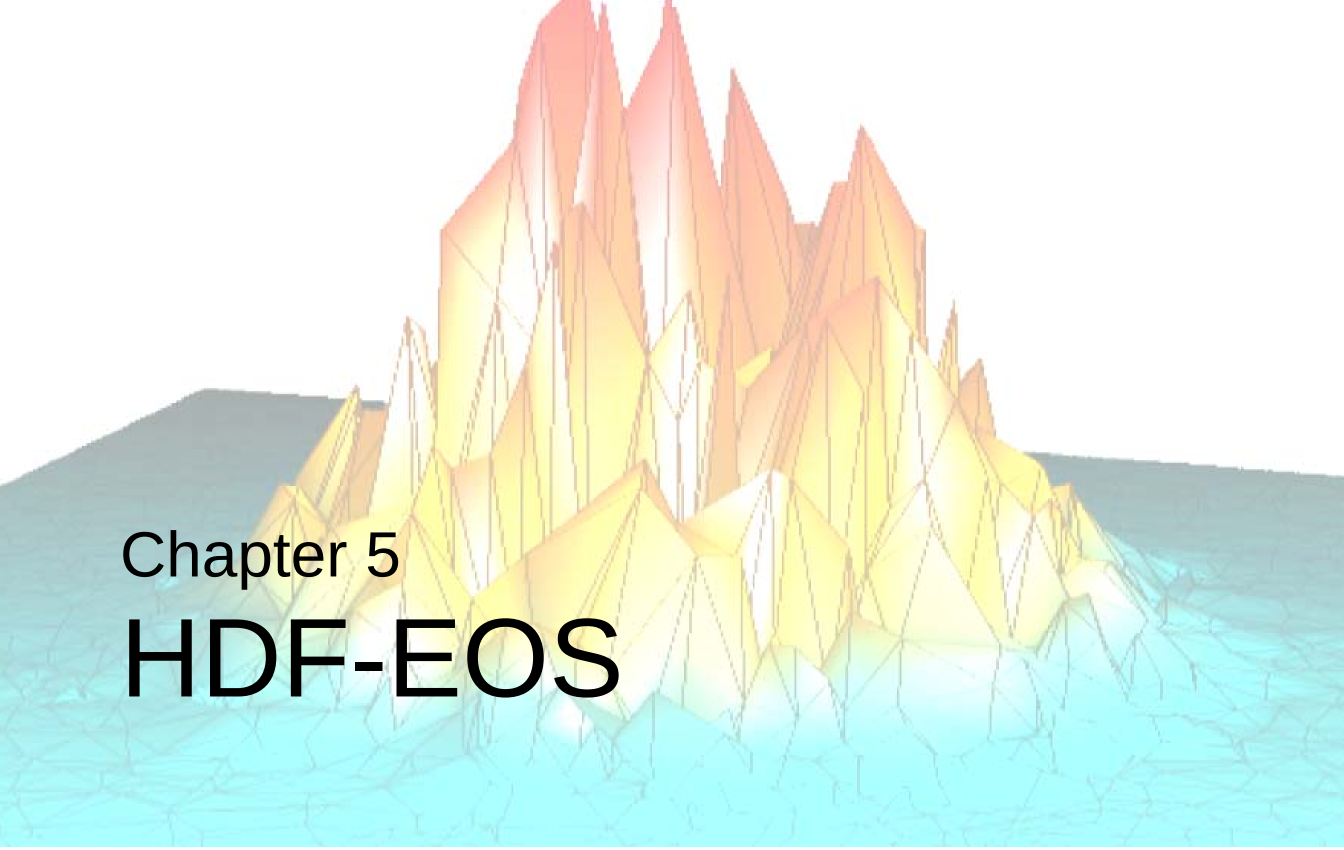
```
OUTNAME STRING = 'My Name'
OUTCLASS STRING = 'My VGroup Class'
```

Version History

4.0	Introduced

See Also

HDF_VG_GETINFO



Chapter 5

HDF-EOS

The following topics are covered in this appendix:

Overview of the HDF-EOS

HDF-EOS (Hierarchical Data Format-Earth Observing System) is an extension of NCSA (National Center for Supercomputing Applications) HDF and uses HDF calls as an underlying basis. This API contains functionality for creating, accessing and manipulating Grid, Point and Swath structures. IDL's HDF-EOS routines all begin with the prefix "EOS_". This version of IDL supports HDF-EOS 2.8.

Note

On the AIX platform, the HDF-EOS library supports version 2.4.

HDF-EOS is a product of NASA, information may be found at:

```
http://hdfeos.gsfc.nasa.gov
```

Feature Routines

HDF-EOS is an extension of NCSA (National Center for Supercomputing Applications) HDF and uses HDF calls as an underlying basis. This API contains functionality for creating, accessing and manipulating Grid, Point and Swath structures.

The Grid interface is designed to support data that has been stored in a rectilinear array based on a well-defined and explicitly supported projection.

Tips on writing a grid:

- Setting a compression method affects all subsequently defined fields
- Setting a tiling scheme affects all subsequently defined fields

The Point interface is designed to support data that has associated geolocation information, but is not organized in any well-defined spatial or temporal way.

Tips on writing a point:

- Every level in a point data set must be linked into the hierarchy
- Before two levels can be linked, a link field must exist

The Swath interface is tailored to support time-oriented data such as satellite swaths (which consist of a time-oriented series of scanlines), or profilers (which consist of a time-oriented series of profiles).

Tips on writing a swath:

- Define dimensions before using them to define fields of maps
- Setting a compression method affects all subsequently defined fields
- If a dimension map is not defined, a one-to-one mapping is assumed during subsetting.

HDF-EOS Programming Model

Writing

- open file
- create object
- define structure
- detach object
- attach object
- write data
- detach object
- close file

Reading

- open file
- attach object
- inquire object
- read data
- detach object
- close file

Note

When writing an HDF-EOS object, be sure to detach the object before attaching it for the first time. This will initialize the library for the new object. The object will not be written correctly if the above model is not followed.

Note on Array Ordering

In versions prior to version 5.5, IDL was inconsistent in its handling of array ordering between IDL and C-language routines in the HDF-EOS library. Beginning with IDL 5.5, all data arrays (either input or output) use standard IDL array ordering. Dimension size vectors and dimension name lists are also now in IDL order rather than in C-language order.

Programs written using versions of the IDL HDF-EOS routines prior to IDL version 5.5 may have been created to intentionally compensate for the previous behavior. When used with IDL 5.5 and later versions of the IDL HDF-EOS routines, these programs may generate incorrect results.

Affected routines include:

EOS_SW_DEFDATAFIELD

EOS_SW_DEFGEOFIELD

EOS_SW_EXTRACTPERIOD

EOS_SW_EXTRACTREGION

EOS_SW_PERIODINFO

EOS_SW_READFIELD

EOS_SW_REGIONINFO

EOS_SW_WRITEDATAMETA

EOS_SW_WRITEFIELD

EOS_SW_WRITEGEOMETA

EOS_GD_DEFFIELD

EOS_GD_DEFTILE

EOS_GD_READFIELD

EOS_GD_READTILE

EOS_GD_REGIONINFO

EOS_GD_TILEINFO

EOS_GD_WRITEFIELD

EOS_GD_WRITEFIELDMETA

EOS_GD_WRITETILE

Note

For the EOS_GD_READFIELD, EOS_SW_READFIELD, EOS_GD_WRITEFIELD, and EOS_SW_WRITEFIELD routines, the START, STRIDE, and EDGE keywords should also be specified in the IDL dimension order.

Note

EOS_GD_INQDIMS and EOS_SW_INQDIMS return dimension size and name information without consideration of order.

Alphabetical Listing of EOS Routines

EOS_EH_CONVANG

EOS_EH_GETVERSION

EOS_EH_IDINFO

EOS_EXISTS

EOS_GD_ATTACH

EOS_GD_ATTRINFO

EOS_GD_BLKSOMOFFSET

EOS_GD_CLOSE

EOS_GD_COMPINFO

EOS_GD_CREATE

EOS_GD_DEFBOXREGION

EOS_GD_DEFCOMP

EOS_GD_DEFDIM

EOS_GD_DEFFIELD

EOS_GD_DEFORIGIN

EOS_GD_DEFPIXREG

EOS_GD_DEFPROJ

EOS_GD_DEFTILE

EOS_GD_DEFVRTREGION

EOS_GD_DETACH

EOS_GD_DIMINFO

EOS_GD_DUPREGION

EOS_GD_EXTRACTREGION

EOS_GD_FIELDINFO

EOS_GD_GETFILLVALUE

EOS_GD_GETPIXELS

EOS_GD_GETPIXVALUES

EOS_GD_GRIDINFO

EOS_GD_INQATTRS

EOS_GD_INQDIMS

EOS_GD_INQFIELDS

EOS_GD_INQGRID

EOS_GD_INTERPOLATE

EOS_GD_NENTRIES

EOS_GD_OPEN

EOS_GD_ORIGININFO

EOS_GD_PIXREGINFO

EOS_GD_PROJINFO

EOS_GD_QUERY

EOS_GD_READATTR

EOS_GD_READFIELD

EOS_GD_READTILE

EOS_GD_REGIONINFO

EOS_GD_SETFILLVALUE

EOS_GD_SETTILECACHE

EOS_GD_TILEINFO

EOS_GD_WRITEATTR

EOS_GD_WRITEFIELD

EOS_GD_WRITEFIELDMETA

EOS_GD_WRITETILE

EOS_PT_ATTACH

EOS_PT_ATTRINFO

EOS_PT_BCKLINKINFO

EOS_PT_CLOSE

EOS_PT_CREATE

EOS_PT_DEFBOXREGION

EOS_PT_DEFLEVEL

EOS_PT_DEFLINKAGE

EOS_PT_DEFTIMEPERIOD

EOS_PT_DEFVRTREGION

EOS_PT_DETACH

EOS_PT_EXTRACTPERIOD

EOS_PT_EXTRACTREGION

EOS_PT_FWDLINKINFO

EOS_PT_GETLEVELNAME

EOS_PT_GETRECNUMS

EOS_PT_INQATTRS

EOS_PT_INQPOINT

EOS_PT_LEVELINDX

EOS_PT_LEVELINFO

EOS_PT_NFIELDS

EOS_PT_NLEVELS

EOS_PT_NRECS

EOS_PT_OPEN

EOS_PT_PERIODINFO

EOS_PT_PERIODRECS

EOS_PT_QUERY

EOS_PT_READATTR

EOS_PT_READLEVEL

EOS_PT_REGIONINFO

EOS_PT_REGIONRECS

EOS_PT_SIZEOF

EOS_PT_UPDATELEVEL

EOS_PT_WRITEATTR

EOS_PT_WRITELEVEL

EOS_QUERY

EOS_SW_ATTACH

EOS_SW_ATTRINFO

EOS_SW_CLOSE

EOS_SW_COMPINFO

EOS_SW_DEFBOXREGION

EOS_SW_DEFBOXREGION

EOS_SW_DEFCOMP

EOS_SW_DEFDATAFIELD

EOS_SW_DEFDIM

EOS_SW_DEFDIMMAP

EOS_SW_DEFGEOFIELD

EOS_SW_DEFIDXMAP

EOS_SW_DEFTIMEPERIOD

EOS_SW_DEFVRTREGION

EOS_SW_DETACH

EOS_SW_DIMINFO

EOS_SW_DUPREGION

EOS_SW_EXTRACTPERIOD

EOS_SW_EXTRACTREGION

EOS_SW_FIELDINFO

EOS_SW_GETFILLVALUE

EOS_SW_IDXMAPINFO

EOS_SW_INQATTRS

EOS_SW_INQDATAFIELDS

EOS_SW_INQDIMS

EOS_SW_INQGEOFIELDS

EOS_SW_INQIDXMAPS

EOS_SW_INQMAPS

EOS_SW_INQSWATH

EOS_SW_MAPINFO

EOS_SW_NENTRIES

EOS_SW_OPEN

EOS_SW_PERIODINFO

EOS_SW_QUERY

EOS_SW_READATTR

EOS_SW_READFIELD

EOS_SW_REGIONINFO

EOS_SW_SETFILLVALUE

EOS_SW_WRITEATTR

EOS_SW_WRITEDATAMETA

EOS_SW_WRITEFIELD

EOS_SW_WRITEGEOMETA

EOS_EH_CONVANG

This function converts angles between three units: decimal degrees, radians, and packed degrees-minutes-seconds. In the degrees-minutes-seconds unit, an angle is expressed as an integral number of degrees and minutes and a float point value of seconds packed as a single double as follows: DDDMMMSSS.SS.

Syntax

Result = EOS_EH_CONVANG(*inAngle*, *code*)

Return Value

Returns angle in desired units.

Arguments

inAngle

Input angle (float).

code

Conversion code (long). Allowable values are:

- 0 = Radians to Degrees
- 1 = Degrees to Radians
- 2 = DMS to Degrees
- 3 = Degrees to DMS
- 4 = Radians to DMS
- 5 = DMS to Radians

Keywords

None

Examples

To convert 27.5 degrees to packed format:

```
inAng = 27.5
outAng = EOS_EH_CONVANG(inAng, 3)
```

outAng will contain the value 27030000.00.

Version History

5.2	Introduced

EOS_EH_GETVERSION

The EOS_EH_GETVERSION function is used to retrieve the HDF-EOS version string of an HDF-EOS file, which is returned in the *version* argument. This designates the version of HDF-EOS that was used to create the file. This string is of the form “HDFEOS_Vmaj.min” where maj is the major version and min is the minor version.

Syntax

Result = EOS_EH_GETVERSION(*fid*, *version*)

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise.

Arguments

fid

File id (long) returned by EOS_SW_OPEN, EOS_GD_OPEN, or EOS_PT_OPEN.

version

HDF-EOS version (string).

Keywords

None

Examples

To get the HDF-EOS version (assumed to be 2.3) used to create the HDF-EOS file:

```
fid = EOS_SW_OPEN("Swathfile.hdf", /READ)
status = EOS_EH_GETVERSION(fid, version)
```

`version` will contain the string “HDFEOS_V2.3”.

Version History

5.2	Introduced

EOS_EH_IDINFO

This function returns the HDF file IDs corresponding to the HDF-EOS file ID returned by EOS_SW_OPEN, EOS_GD_OPEN, or EOS_PT_OPEN. These IDs can then be used to create or access native HDF structures such as SDS arrays, Vdatas, or HDF attributes within an HDF-EOS file.

Syntax

Result = EOS_EH_IDINFO(*fid*, *HDFfid*, *sdInterfaceID*)

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise.

Arguments

fid

File ID (long) returned by EOS_SW_OPEN, EOS_GD_OPEN, or EOS_PT_OPEN.

HDFfid

A named variable that will contain the HDF file ID (long) returned by Hopen.

sdInterfaceID

A named variable that will contain the SD interface ID (long) returned by SDstart.

Keywords

None

Examples

To create a vdata within an existing HDF-EOS file:

```
fid = EOS_SW_OPEN("SwathFile.hdf", /RDWR)
status = EOS_EH_IDINFO(fid, hdffid, sdid)
```

Version History

EOS_EXISTS

The EOS_EXISTS function determines whether the current HDF-EOS extensions are supported on the current platform.

Syntax

Result = EOS_EXISTS( )

Return Value

Returns success (1) if the HDF-EOS extensions are supported, and fail (0) if not.

Arguments

None

Keywords

None

Examples

```
IF (~ HDF_EOS_EXISTS) THEN PRINT,'HDF-EOS not available.'
```

Version History

5.2.1	Introduced

EOS_GD_ATTACH

This function attaches to the grid using the gridname parameter as the identifier.

Syntax

Result = EOS_GD_ATTACH(*fid*, *gridname*)

Return Value

Returns the grid handle (gridID) if successful and FAIL(–1) otherwise.

Arguments

fid

Grid file id (long) returned by EOS_GD_OPEN.

gridname

Name of grid (string) to be attached.

Keywords

None

Examples

In this example, we attach to the previously created grid, "ExampleGrid", within the HDF file, GridFile.hdf, referred to by the handle, fid:

```
gridID = EOS_GD_ATTACH(fid, "ExampleGrid")
```

The grid can then be referenced by subsequent routines using the handle, gridID.

Version History

5.2	Introduced

See Also

EOS_GD_DETACH

EOS_GD_ATTRINFO

This function returns number type and number of elements (count) of a grid attribute.

Syntax

Result = EOS_GD_ATTRINFO(*gridID*, *attrname*, *numbertype*, *count*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

attrname

Attribute name (string).

numbertype

A named variable that will contain the number type (long) of an attribute.

count

A named variable that will contain the number of total bytes in an attribute (long).

Keywords

None

Examples

In this example, we return information about the ScalarFloat attribute:

```
status = EOS_GD_ATTRINFO(pointID, "ScalarFloat", nt, count)
```

Version History

5.2	Introduced

EOS_GD_BLKSOMOFFSET

This function writes block SOM offset values. This is a special function for SOM MISR data.

Syntax

Result = EOS_GD_BLKSOMOFFSET(*gridID*, *offset*, *code*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

gridID

The grid ID (long) as returned from EOS_GD_ATTACH.

offset

A scalar or array of offset values. The offset must be initialized to the correct data type and number of elements for the values to be written correctly.

code

The type of action performed (read (r), write (w)). This value must be set to either the string r or w. If the string value is not recognized, the *code* defaults to r.

Keywords

None

Version History

5.3	Introduced

EOS_GD_CLOSE

This function closes the HDF grid file.

Syntax

Result = EOS_GD_CLOSE(*fid*)

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise.

Arguments

fid

Grid file id (long) returned by EOS_GD_OPEN.

Keywords

None

Examples

```
status = EOS_GD_CLOSE(fid)
```

Version History

5.2	Introduced

EOS_GD_COMPINFO

This function returns the compression code and compression parameters for a given field.

Syntax

Result = EOS_GD_COMPINFO(*gridID*, *fieldname*, *compcode*, *compparm*)

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

fieldname

Fieldname (string).

compcode

A named variable that will contain the HDF compression code (long).

compparm

A named variable that will contain the compression parameters (long array).

Keywords

None

Examples

To retrieve the compression information about the Opacity field defined in the EOS_GD_DEFCOMP section:

```
status = EOS_GD_COMPINFO(gridID, "Opacity", compcode,
compparm)
```

Version History

5.2	Introduced

See Also

EOS_GD_DEFCOMP

EOS_GD_CREATE

This function creates a grid within the file. The grid is created as a Vgroup within the HDF file with the name gridname and class GRID. This function establishes the resolution of the grid, (i.e., the number of rows and columns), and its location within the complete global projection through the upleftpt and lowrightpt arrays. These arrays should be in meters for all GCTP projections other than the Geographic Projection, which should be in packed degree format (q.v. below). For GCTP projection information, see the *HDF-EOS User's Guide, Volume 2: Reference Guide* provided by NASA.

Syntax

Result = EOS_GD_CREATE(*fid*, *gridname*, *xdimsize*, *ydimsize*, *upleftpt*, *lowrightpt*)

Return Value

Returns the grid handle (gridID) and FAIL(–1) otherwise.

Arguments

fid

Grid file id (long) returned by EOS_GD_OPEN.

gridname

Name of grid (string) to be created.

xdimsize

Number of columns (long) in grid.

ydimsize

Number of rows (long) in grid.

upleftpt

Location (double, 2 element array) of upper left corner of the upper left pixel.

lowrightpt

Location (double, 2 element array) of lower right corner of the lower right pixel.

Keywords

None

Examples

In this example, we create a UTM grid bounded by 54 E to 60 E longitude and 20 N to 30 N latitude. We divide it into 120 bins along the x-axis and 200 bins along the y-axis.

```
uplft[0]=10584.50041d
uplft[1]=3322395.95445d
lowrgt[0]=813931.10959d
lowrgt[1]=214162.53278d
xdim=120
ydim=200
gridID = EOS_GD_CREATE(fid, "UTMGrid", xdim, ydim, uplft, lowrgt)
```

The grid structure is then referenced by subsequent routines using the handle, gridID.

The xdim and ydim values are referenced in the field definition routines by the reserved dimensions: XDim and YDim.

For the Polar Stereographic, Goode Homolosine and Lambert Azimuthal projections, we have established default values in the case of an entire hemisphere for the first projection, the entire globe for the second and the entire polar or equatorial projection for the third.

In the case of the Geographic projection (linear scale in both longitude latitude), the upleftpt and lowrightpt arrays contain the longitude and latitude of these points in packed degree format (DDDMMMSSS.SS).

- upleftpt- Array that contains the X-Y coordinates of the upper left corner of the upper left pixel of the grid. First and second elements of the array contain the X and Y coordinates respectively. The upper left X coordinate value should be the lowest X value of the grid. The upper left Y coordinate value should be the highest Y value of the grid.
- lowrightpt - Array that contains the X-Y coordinates of the lower right corner of the lower right pixel of the grid. First and second elements of the array contain the X and Y coordinates respectively. The lower right X coordinate

value should be the highest X value of the grid. The lower right Y coordinate value should be the lowest Y value of the grid.

If the projection is geographic (i.e., projcode=0) then the X-Y coordinates should be specified in degrees/minutes/seconds (DDDMMMSSS.SS) format. The first element of the array holds the longitude and the second element holds the latitude. Latitudes are from –90 to +90 and longitudes are from –180 to +180 (west is negative).

For all other projection types the X-Y coordinates should be in meters in double precision. These coordinates have to be computed using the GCTP software with the same projection parameters that have been specified in the projparm array. For UTM projections use the same zone code and its sign (positive or negative) while computing both upper left and lower right corner X-Y coordinates irrespective of the hemisphere.

Version History

5.2	Introduced

EOS_GD_DEFBOXREGION

This function defines a longitude-latitude box region for a grid. It returns a grid region ID which is used by the EOS_GD_EXTRACTREGION function to read all the entries of a data field within the region.

Syntax

Result = EOS_GD_DEFBOXREGION(*gridID*, *cornerlon*, *cornerlat*)

Return Value

Returns the grid region ID if successful and FAIL (–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

cornerlon

Longitude (double, 2 element array) in decimal degrees of box corners.

cornerlat

Latitude (double, 2 element array) in decimal degrees of box corners.

Keywords

None

Examples

In this example, we define the region to be the first quadrant of the Northern hemisphere:

```
cornerlon[0] = 0.d
cornerlat[0] = 90.d
cornerlon[1] = 90.d
cornerlat[1] = 0.d
regionID = EOS_GD_DEFBOXREGION(EOS_GD_id, cornerlon, cornerlat)
```

Version History

5.2	Introduced

EOS_GD_DEFCOMP

This function sets the HDF field compression for subsequent grid field definitions. The compression does not apply to one-dimensional fields. The compression schemes currently supported are: run length encoding (1), skipping Huffman (3), deflate (gzip) (4) and no compression (0, the default). Deflate compression requires a single integer compression parameter in the range of one to nine with higher values corresponding to greater compression.

Compressed fields are written using the standard EOS_GD_WRITEFIELD function, however, the entire field must be written in a single call. If this is not possible, the user should consider tiling. See EOS_GD_DEFTILE for further information. Any portion of a compressed field can then be accessed with the EOS_GD_READFIELD function. Compression takes precedence over merging so that multi-dimensional fields that are compressed are not merged. The user should refer to the HDF Reference Manual for a fuller explanation of compression schemes and parameters.

Syntax

Result = EOS_GD_DEFCOMP(*gridID*, *compcode* [, *compparm*])

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

compcode

HDF compression code (long).

Allowable values are:

- 0 = None
- 1 = Run Length Encoding (RLE)
- 3 = Skipping Huffman
- 4 = Deflate (gzip)

compparm

Compression parameters array. Compparm is an array argument whose value(s) depend on the compression scheme selected.

Keywords

None

Examples

Suppose we wish to compress the Pressure field using run length encoding, the Opacity field using deflate compression, the Spectra field with skipping Huffman compression, and use no compression for the Temperature field:

```
status = EOS_GD_DEFCOMP(gridID, 1)
status = EOS_GD_DEFFIELD(gridID, "Pressure", "YDim,XDim", 5)
compparm[0] = 5
status = EOS_GD_DEFCOMP(gridID, 4, compparm)
status = EOS_GD_DEFFIELD(gridID, "Opacity", "YDim,XDim", 5)
status = EOS_GD_DEFCOMP(gridID, 3)
status = EOS_GD_DEFFIELD(gridID, "Spectra", "Bands,YDim,XDim", 5)
status = EOS_GD_DEFCOMP(gridID, 0)
status = EOS_GD_DEFFIELD(gridID, "Temperature", "YDim,XDim", 5,$
   /MERGE)
```

Note that the MERGE keyword will be ignored in the Temperature field definition.

Version History

5.2	Introduced

EOS_GD_DEFDIM

This function defines dimensions that are used by the field definition routines (described subsequently) to establish the size of the field.

Syntax

Result = EOS_GD_DEFDIM(*gridID*, *dimname*, *dim*)

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

dimname

Name of dimension to be defined (string).

dim

The size of the dimension (long).

Keywords

None

Examples

In this example, we define a dimension, Band, with size 15:.

```
status = EOS_GD_DEFDIM(gridID, "Band", 15)
```

To specify an unlimited dimension that can be used to define an appendable array, the dimension value should be set to zero:

```
status = EOS_GD_DEFDIM(gridID, "Unlim", 0)
```

Version History

5.2	Introduced

EOS_GD_DEFFIELD

This function defines data fields to be stored in the grid. The dimensions are entered as a string consisting of geolocation dimensions separated by commas. The API will attempt to merge into a single object those fields that share dimensions and in case of multidimensional fields, numbertype. If the MERGE keyword is not set, the API will not attempt to merge it with other fields. Fields using the unlimited dimension will not be merged. Because merging breaks the one-to-one correspondence between HDF-EOS fields and HDF SDS arrays, it should not be set if the user wishes to access the HDF-EOS fields directly using HDF.

Note

Array ordering of variables used or returned by this routine changed in IDL 5.5. Programs written for versions of this routine prior to IDL 5.5 may need to be modified to work correctly with the current version. See "Note on Array Ordering" on page 579 for details.

Syntax

Result = EOS_GD_DEFFIELD(*gridID*, *fieldname*, *dimlist*, *numbertype* [, /MERGE])

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

fieldname

Name of field (string) to be defined.

dimlist

The list of data dimensions (string) defining the field.

numbertype

The HDF data type (long) of the data stored in the field

Keywords

MERGE

If set, automerge will occur.

Examples

In this example, we define a grid field, Temperature with dimensions XDim and YDim (as established by the EOS_GD_CREATE routine) containing 4-byte floating point numbers and a field, Spectra, with dimensions XDim, YDim, and Bands:

```
status = EOS_GD_DEFFIELD(gridID, "Temperature", &
   "YDim,XDim", 5, /MERGE)
status = EOS_GD_DEFFIELD(gridID, "Spectra", "Bands,YDim,XDim", 5)
```

Version History

5.2	Introduced

EOS_GD_DEFORIGIN

This function defines the origin of the grid data. This allows the user to select any corner of the grid as the origin.

Syntax

Result = EOS_GD_DEFORIGIN(*gridID*, *origincode*)

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

origincode

Location of the origin (long) of the grid data. The allowable values are:

- 0 = Upper left
- 1 = Upper right
- 2 = Lower left
- 3 = Lower right

Keywords

None

Examples

In this Example we define the origin of the grid to be the Lower Right corner:

```
status = EOS_GD_DEFORIGIN(gridID, 3)
```

Version History

5.2	Introduced

EOS_GD_DEFPIXREG

This function defines whether the pixel center or pixel corner (as defined by the EOS_GD_DEFORIGIN function) is used when requesting the location (longitude and latitude) of a given pixel.

Syntax

Result = EOS_GD_DEFPIXREG(*gridID*, *pixreg*)

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

pixreg

Pixel registration (long). The allowable values are:

- 0 = Center
- 1 = Corner

Keywords

None

Examples

In this example, we define the pixel registration to be the corner of the pixel cell:

```
status = EOS_GD_DEFPIXREG(gridID, 1)
```

Version History

5.2	Introduced

EOS_GD_DEFPROJ

This function defines the GCTP projection and projection parameters of the grid. For GCTP projection information, see the *HDF-EOS User's Guide, Volume 2: Reference Guide* provided by NASA.

Syntax

Result = EOS_GD_DEFPROJ(*gridID*, *projcode*, *zonecode*, *spherecode*, *projparm*)

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

projcode

GCTP projection code (long).

zonecode

GCTP zone code used by UTM projection (long).

spherecode

GCTP spheroid code (long).

projparm

GCTP projection parameter array.

Keywords

None

Examples

Example 1

In this example, we define a Universal Transverse Mercator (UTM) grid bounded by 54 E - 60 E longitude and 20 N - 30 N latitude - UTM zonecode 40, using default spheroid (Clarke 1866), spherecode = 0:

```
spherecode = 0
zonecode = 40
status = EOS_GD_DEFPROJ(gridID, 1, zonecode, spherecode, 0)
```

Example 2

In this example, we define a Polar Stereographic projection of the Northern Hemisphere (True scale at 90 N, 0 Longitude below pole) using the International 1967 spheroid:

```
spherecode = 3
projparm = lonarr (13)
;Set Long below pole & true scale in DDDMMMSSS.SSS form
projparm[5] = 90000000.00
status = EOS_GD_DEFPROJ(gridID, 6, 0, spherecode, projparm)
```

Example 3

Finally, we define a Geographic projection. In this case, neither the zone code, sphere code, or the projection parameters are used:

```
status = EOS_GD_DEFPROJ(gridID, 0, 0, 0, 0)
```

Version History

5.2	Introduced

EOS_GD_DEFTILE

This function defines the tiling dimensions for fields defined following this function call, analogous to the procedure for setting the field compression scheme using EOS_GD_DEFCOMP. The number of tile dimensions and subsequent field dimensions must be the same and the tile dimensions must be integral divisors of the corresponding field dimensions. A tile dimension set to 0 will be equivalent to 1.

Note

Array ordering of variables used or returned by this routine changed in IDL 5.5. Programs written for versions of this routine prior to IDL 5.5 may need to be modified to work correctly with the current version. See "Note on Array Ordering" on page 579 for details.

Syntax

Result = EOS_GD_DEFTILE(*gridID*, *tilecode* [, *tilerank*, *tiledims*])

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

tilecode

Tile code (long): 0 = notile, 1 = tile

tilerank

The number of tile dimensions (long) (optional).

tiledims

Tile dimensions (long) (optional).

Keywords

None

Examples

We will define four fields in a grid, two 2-D fields of the same size with the same tiling, a three-dimensional field with a different tiling scheme, and a fourth with no tiling. We assume that XDim is 200 and YDim is 300.

```
tiledims[0] = 100
tiledims[1] = 200
status = EOS_GD_DEFTILE(gridID, 1, 2, tiledims)
status = EOS_GD_DEFFIELD(gridID, "Pressure", "YDim,XDim", 22)
status = EOS_GD_DEFFIELD(gridID, "Temperature", "YDim,XDim", 5)
tiledims[0] = 1
tiledims[1] = 150
tiledims[2] = 100
status = EOS_GD_DEFTILE(gridID, 1, 3, tiledims)
status = EOS_GD_DEFFIELD(gridID, "Spectra", "Bands,YDim,XDim", 5)
status = EOS_GD_DEFTILE(gridID, 0, 0)
status = EOS_GD_DEFFIELD(gridID, "Communities", "YDim,XDim", 24,
/MERGE)
```

Version History

5.2	Introduced

EOS_GD_DEFVRTREGION

This function subsets on a monotonic field or contiguous elements of a dimension. Whereas the EOS_GD_DEFBOXREGION function subsets along the XDim and YDim dimensions, this function allows the user to subset along any other dimension. The region is specified by a set of minimum and maximum values and can represent either a dimension index (case 1) or field value range (case 2). In the second case, the field must be one-dimensional and the values must be monotonic (strictly increasing or decreasing) in order that the resulting dimension index range be contiguous. (For the current version of this routine, the second option is restricted to fields with number type: 22, 24, 5, 6)

This function may be called after EOS_GD_DEFBOXREGION to provide both geographic and “vertical” subsetting. In this case the user provides the id from the previous subset call. (This same id is then returned by the function.) This routine may also be called “stand-alone” by setting the input id to (–1).

This function may be called up to eight times with the same region ID. It this way a region can be subsetted along a number of dimensions.

The EOS_GD_REGIONINFO and EOS_GD_EXTRACTREGION functions work as before, however the field to be subsetted, (the field specified in the call to EOS_GD_REGIONINFO and EOS_GD_EXTRACTREGION) must contain the dimension used explicitly in the call to EOS_GD_DEFVRTREGION (case 1) or the dimension of the one-dimensional field (case 2).

Syntax

Result = EOS_GD_DEFVRTREGION(*gridID*, *regionID*, *vertObj*, *range*)

Return Value

Returns the grid region ID if successful and FAIL (–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

regionID

Region or period id (long) from previous subset call or –1 to start a new region.

vertObj

Dimension or field to subset (string).

range

Minimum and maximum range for subset (double) 2 element array.

Keywords

None

Examples

Suppose we have a field called Pressure of dimension Height whose values increase from 100 to1000.   If we desire all the elements with values between 500 and 800, we make the call:

```
range[0] = 500.d
range[1] = 800.d
regionID = EOS_GD_DEFVRTREGION(gridID, -1, "Pressure", range)
```

The routine determines the elements in the Height dimension which correspond to the values of the Pressure field between 500 and 800.

If we wish to specify the subset as elements 2 through 5 (0 - based) of the Height dimension, the call would be:

```
range[0] = 2.d
range[1] = 5.d
regionID = EOS_GD_DEFVRTREGION(gridID, -1, "DIM:Height", range)
```

The “DIM:” prefix tells the routine that the range corresponds to elements of a dimension rather than values of a field.

If a previous subset region or period was defined with id, subsetID, that we wish to refine further with the vertical subsetting defined above we make the call:

```
regionID = EOS_GD_DEFVRTREGION(gridID, subsetID, "Pressure", $
   range)
```

The return value, regionID, is set equal to subsetID. That is, the subset region is modified rather than a new one created.

In this example, any field to be subsetted must contain the Height dimension.

Version History

5.2	Introduced

EOS_GD_DETACH

This function detaches from the grid interface. This routine should be run before exiting from the grid file for every grid opened by EOS_GD_CREATE or EOS_GD_ATTACH.

Syntax

Result = EOS_GD_DETACH(*gridID*)

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

Keywords

None

Examples

In this example, we detach a grid structure:

```
status = EOS_GD_DETACH(gridID)
```

Version History

5.2	Introduced

See Also

EOS_GD_ATTACH

EOS_GD_DIMINFO

This function retrieves the size of the specified dimension.

Syntax

Result = EOS_GD_DIMINFO(*gridID*, *dimname*)

Return Value

Size of dimension. If FAIL(–1), could signify an improper grid id or dimension name.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

dimname

Dimension name (string)

Keywords

None

Examples

In this example, we retrieve information about the dimension, “Bands”:

```
dimsize = EOS_GD_DIMINFO(gridID, "Bands")
```

Version History

5.2	Introduced

EOS_GD_DUPREGION

This function copies the information stored in a current region or period to a new region or period and generates a new id. It is useful when the user wishes to further subset a region (period) in multiple ways.

Syntax

Result = EOS_GD_DUPREGION(*regionID*)

Return Value

Returns new region or period ID or FAIL (–1) if the region cannot be duplicated.

Arguments

regionID

Region or period id (long) returned by EOS_GD_DEFBOXREGION or EOS_GD_DEFVRTREGION.

Keywords

None

Examples

In this example, we first subset a grid with EOS_GD_DEFBOXREGION, duplicate the region creating a new region ID, regionID2, and then perform two different vertical subsets of these (identical) geographic subset regions:

```
regionID = EOS_GD_DEFBOXREGION(gridID, cornerlon, cornerlat)
regionID2 = EOS_GD_DUPREGION(regionID)
regionID = EOS_GD_DEFVRTREGION(gridID, regionID, "Pressure",$
   rangePres)
regionID2 = EOS_GD_DEFVRTREGION(gridID, regionID2, $
   "Temperature", rangeTemp)
```

Version History

5.2	Introduced

EOS_GD_EXTRACTREGION

This function reads data into the data buffer from a subsetted region as defined by EOS_GD_DEFBOXREGION.

Syntax

Result = EOS_GD_EXTRACTREGION(*gridID*, *regionID*, *fieldname*, *buffer*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

regionID

Region or period id (long) returned by EOS_GD_DEFBOXREGION.

fieldname

Field to subset (string).

buffer

A named variable that will contain the data Buffer.

Keywords

None

Examples

In this example, we extract data from the “Temperature” field from the region defined in EOS_GD_DEFBOXREGION. The size of the subsetted region for the field is given by the EOS_GD_REGIONINFO routine.

```
status = EOS_GD_EXTRACTREGION(EOS_GD_id, regionID, $
   "Temperature", datbuf32)
```

Version History

5.2	Introduced

EOS_GD_FIELDINFO

This function retrieves information on a specific data field.

Syntax

Result = EOS_GD_FIELDINFO(*gridID*, *fieldname*, *rank*, *dims*, *numbertype*, *dimlist*)

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) if the specified field does not exist.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

fieldname

Fieldname (string).

rank

A named variable that will contain the pointer to rank (long) of the field.

dims

A named variable that will contain an array (long) of the dimension sizes of the field.

numbertype

A named variable that will contain the HDF data type (long) of the field.

dimlist

A named variable that will contain the dimension list (string).

Keywords

None

Examples

In this example, we retrieve information about the Spectra data fields:

```
status = EOS_GD_FIELDINFO(gridID, "Spectra", rank, dims,$
   numbertype, dimlist)
```

Version History

5.2	Introduced

EOS_GD_GETFILLVALUE

This function retrieves the fill value for the specified field.

Syntax

Result = EOS_GD_GETFILLVALUE(*gridID*, *fieldname*, *fillvalue*)

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

fieldname

Fieldname (string).

fillvalue

A named variable that will contain the fill value.

Keywords

None

Examples

In this example, we get the fill value for the “Temperature” field:

```
status = EOS_GD_GETFILLVALUE(gridID, "Temperature", tempfill)
```

Version History

5.2	Introduced

EOS_GD_GETPIXELS

This function returns the pixel rows and columns for specified longitude/latitude pairs. This function converts longitude/latitude pairs into (0-based) pixel rows and columns. The origin is the upper left-hand corner of the grid. This routine is the pixel subsetting equivalent of EOS_GD_DEFBOXREGION.

Syntax

Result = EOS_GD_GETPIXELS(*gridID*, *nLonLat*, *lonVal*, *latVal*, *pixRow*, *pixCol*)

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

nLonLat

Number of longitude/latitude pairs (long).

lonVal

Longitude values in degrees (double, 1D array).

latVal

Latitude values in degrees (double, 1D array).

pixRow

A named variable that will contain the pixel Rows (long array).

pixCol

A named variable that will contain the pixel Columns (long array).

Keywords

None

Examples

This example converts two pairs of longitude/latitude values to rows and columns. The rows and columns of the two pairs will be returned in the rowArr and colArr arrays:

```
lonArr[0] = 134.2d
latArr[0] = -20.8d
lonArr[1] = 15.8d
latArr[1] = 84.6d
status = EOS_GD_GETPIXELS(gridID, 2, lonArr, latArr, rowArr, $
   colArr)
```

Version History

5.2	Introduced

EOS_GD_GETPIXVALUES

This function reads data from a data field for the specified pixels. It is the pixel subsetting equivalent of EOS_GD_EXTRACTREGION. All entries along the non-geographic dimensions (i.e., NOT XDim and YDim) are returned.

Syntax

Result = EOS_GD_GETPIXVALUES(*gridID*, *nPixels*, *pixCol*, *pixRow*, *fieldname*, *buffer*)

Return Value

Returns size of data buffer if successful and FAIL(–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH

nPixels

Number of pixels (long).

pixCol

Pixel Columns (long array).

pixRow

Pixel Rows (long array).

fieldname

Field (string) from which to extract data values.

buffer

A named variable that will contain data values.

Keywords

None

Examples

To read values from the Spectra field with dimensions, Bands, YDim, and XDim:

```
bufsiz = EOS_GD_GETPIXVALUES(gridID, 2, rowArr, colArr, $
   "Spectra", buffer)
```

Version History

5.2	Introduced

EOS_GD_GRIDINFO

This function returns the number of rows, columns and the location, in meters, of the upper left and lower right corners of the grid image.

Syntax

Result = EOS_GD_GRIDINFO(*gridID*, *xdimsize*, *ydimsize*, *upleft*, *lowright*)

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

xdimsize

A named variable that will contain the number of columns in grid (long).

ydimsize

A named variable that will contain the number of rows in grid (long).

upleft

A named variable that will contain the location (double, 2 element array; in meters) of upper left corner.

lowright

A named variable that will contain the location (double, 2 element array; in meters) of lower right corner.

Keywords

None

Examples

In this example, we retrieve information from a previously created grid with a call to EOS_GD_ATTACH:

```
status = EOS_GD_GRIDINFO(gridID, xdimsize, ydimsize, $
   upleft, lowrgt)
```

Version History

5.2	Introduced

EOS_GD_INQATTRS

This function retrieves information about attributes defined in a grid. The attribute list is returned as a string with each attribute name separated by commas.

Note

See STRSPLIT to separate the attribute list.

Syntax

Result = EOS_GD_INQATTRS(*gridID*, *attrlist* [, LENGTH=*variable*])

Return Value

Number of attributes found or (–1) if failure.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

attrlist

A named variable that will contain the attribute list (string) with entries separated by commas.

Keywords

LENGTH

Set this keyword to a named variable that will contain the length of the attribute list as a long integer.

Examples

In this example, we retrieve information about the attributes defined in a grid structure:

```
nattr = EOS_GD_INQATTRS(gridID, attrlist)
```

Version History

5.2	Introduced

EOS_GD_INQDIMS

This function retrieves information about dimensions defined in a grid. The dimension list is returned as a string with each dimension name separated by commas.

Note

See STRSPLIT to separate the dimension list.

Syntax

Result = EOS_GD_INQDIMS(*gridID*, *dimname*, *dims*)

Return Value

Number of dimension entries found. If FAIL(–1), could signify an improper grid id.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

dimname

A named variable that will contain the dimension list (string) with entries separated by commas.

dims

A named variable that will contain an array (long) of the size of each dimension.

Keywords

None

Examples

To retrieve information about the dimensions, use the following statement:

```
ndim = EOS_GD_INQDIMS(gridID, dimname, dims)
```

Version History

5.2	Introduced

EOS_GD_INQFIELDS

This function retrieves information about the data fields defined in grid. The field list is returned as a string with each data field separated by commas. The rank and numbertype arrays will have an entry for each field.

Note

See STRSPLIT to separate the field list.

Syntax

Result = EOS_GD_INQFIELDS(*gridID*, *fieldlist*, *rank*, *numbertype*)

Return Value

Number of data fields found. If FAIL(–1), could signify an improper grid id.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

fieldlist

A named variable that will contain the listing of data fields (string) with entries separated by commas.

rank

A named variable that will contain the array (long) containing the rank of each data field.

numbertype

A named variable that will contain the array (long) containing the numbertype of each data field.

Keywords

None

Examples

To retrieve information about the data fields, use the following statement:

```
nfld = EOS_GD_INQFIELDS(gridID, fieldlist, rank, numbertype)
```

Version History

5.2	Introduced

EOS_GD_INQGRID

This function retrieves number and names of grids defined in HDF-EOS file. The grid list is returned as a string with each grid name separated by commas.

Note

See STRSPLIT to separate the grid list.

Syntax

Result = EOS_GD_INQGRID(*filename*, *gridlist* [, LENGTH=*variable*])

Return Value

Number of grids found or (–1) if failure.

Arguments

filename

HDF-EOS filename (string).

gridlist

A named variable that will contain the grid list (string) with entries separated by commas.

Keywords

LENGTH

Set this keyword to a named variable that will contain the length of the grid list as a long integer.

Examples

In this example, we retrieve information about the grids defined in an HDF-EOS file, HDFEOS.hdf:

```
ngrid = EOS_GD_INQGRID("HDFEOS.hdf", gridlist)
```

Version History

5.2	Introduced

EOS_GD_INTERPOLATE

This function performs bilinear interpolation on a grid field. It assumes that the pixel data values are uniformly spaced which is strictly true only for an infinitesimally small region of the globe but is a good approximation for a sufficiently small region. The default position of the pixel value is pixel center, however if the pixel registration has been set to 1 (with the EOS_GD_DEFPIXREG function) then the value is located at one of the four corners specified by the EOS_GD_DEFORIGIN routine.

All entries along the non-geographic dimensions (i.e., NOT XDim and YDim) are interpolated and all interpolated values are returned as FLOAT64. The reference for the interpolation algorithm is Numerical Recipes in C (2nd ed). (Note for the current version of this routine, the number type of the field to be interpolated is restricted to 22, 24, 5, 6.)

Syntax

Result = EOS_GD_INTERPOLATE(*gridID*, *Interp*, *lonVal*, *latVal*, *fieldname*, *interpVal*)

Return Value

Returns size in bytes of interpolated data values if successful and FAIL(–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH

nInterp

Number of interpolation points (long).

lonVal

Longitude of interpolation points (double array).

latVal

Latitude of interpolation points (double array).

fieldname

The field (string) from which to interpolate data values.

interpVal

A named variable that will contain the (double) interpolated data values.

Keywords

None

Examples

To interpolate the Spectra field at two geographic data points:

```
lonVal[0] = 134.2d
latVal[0] = -20.8d
lonVal[1] = 15.8d
latVal[1] = 84.6d
bufsiz = EOS_GD_INTERPOLATE(gridID, 2, lonVal, latVal, $
   "Spectra", interpVal)
```

Version History

5.2	Introduced

EOS_GD_NENTRIES

This function returns the number of entries and descriptive string buffer size for a specified entity.

Syntax

Result = EOS_GD_NENTRIES(*gridID*, *entrycode* [, LENGTH=*variable*])

Return Value

Number of entries or FAIL(–1) which could signify an improper grid id or entry code.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

entrycode

Entrycode (long). Allowable values are:

- 0 = Dimensions
- 4 = Datafields

Keywords

LENGTH

Set this keyword to a named variable that will contain the length of the string returned by the corresponding inquiry routine as a long integer.

Examples

In this example, we determine the number of data field entries:

```
ndims = EOS_GD_NENTRIES(gridID, 4)
```

Version History

5.2	Introduced

EOS_GD_OPEN

This function creates a new file or opens an existing one.

Syntax

Result = EOS_GD_OPEN(*filename*, *access* [, /CREATE] [, /RDWR | , /READ])

Return Value

Returns the grid file id handle (fid) if successful and FAIL(–1) otherwise.

Arguments

filename

Complete path and filename (string) for the file to be opened.

Keywords

CREATE

If file exists, delete it, then open a new file for read/write.

RDWR

Open for read/write. If file does not exist, create it.

READ

Open for read only. If file does not exist, error. This is the default.

Examples

In this example, we create a new grid file named, GridFile.hdf. It returns the file handle, fid.

```
fid = EOS_GD_OPEN("GridFile.hdf", /CREATE)
```

Version History

5.2	Introduced

See Also

EOS_GD_CLOSE

EOS_GD_ORIGININFO

This function retrieves the origin code.

Syntax

Result = EOS_GD_ORIGININFO(*gridID*, *origincode*)

Return Value

Returns 0 if successful, and –1 otherwise. A return value of –1 could signify an improper grid id or entry code.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

origincode

A named variable that will contain the origin code (long). See EOS_GD_DEFORIGIN for a list of origin codes and their meanings.

Keywords

None

Version History

5.2	Introduced

EOS_GD_PIXREGINFO

This function retrieves the pixel registration code.

Syntax

Result = EOS_GD_PIXREGINFO(*gridID*, *pixregcode*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

pixregcode

A named variable that will contain the pixel registration code (long).

Keywords

None

Version History

5.2	Introduced

EOS_GD_PROJINFO

This function retrieves the GCTP projection code, zone code, spheroid code and the projection parameters of the grid. For GCTP projection information, see the *HDF-EOS User's Guide, Volume 2: Reference Guide* provided by NASA.

Syntax

Result = EOS_GD_PROJINFO(*gridID*, *projcode*, *zonecode*, *spherecode*, *projparm*)

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

projcode

A named variable that will contain the GCTP projection code (long).

zonecode

A named variable that will contain the GCTP zone code used by UTM projection (long).

spherecode

A named variable that will contain the GCTP spheroid code (long).

projparm

A named variable that will contain the GCTP projection parameter array (double).

Keywords

None

Version History

5.2	Introduced

EOS_GD_QUERY

The EOS_GD_QUERY function returns information about a specified grid.

Syntax

Result = EOS_GD_QUERY(*Filename*, *GridName*, [*Info*])

Return Value

This function returns an integer value of 1 if the file is an HDF file with EOS GRID extensions, and 0 otherwise.

Arguments

Filename

A string containing the name of the file to query.

GridName

A string containing the name of the grid to query.

Info

Returns an anonymous structure containing information about the specified grid. The returned structure contains the following fields:

Field	IDL Data Type	Description
ATTRIBUTES	String array	Array of attribute names
DIMENSION_NAMES	String array	Names of dimensions
DIMENSION_SIZES	Long array	Sizes of dimensions
FIELD_NAMES	String array	Names of fields
FIELD_RANKS	Long array	Ranks (dimensions) of fields
FIELD_TYPES	Long array	IDL types of fields
GCTP_PROJECTION	Long	GCTP projection code

Table 5-1: Fields of the Info Structure

Field	IDL Data Type	Description
GCTP_PROJECTION_PARM	Double array	GCTP projection parameters
GCTP_SPHEROID	Long	GCTP spheroid code
GCTP_ZONE	Long	GCTP zone code (for UTM projection)
IMAGE_LOWRIGHT	Double[2]	Location of lower right corner (meters)
IMAGE_UPLEFT	Double[2]	Location of upper left corner (meters)
IMAGE_X_DIM	Long	Number of columns in grid image
IMAGE_Y_DIM	Long	Number of rows in grid image
NUM_ATTRIBUTES	Long	Number of attributes
NUM_DIMS	Long	Number of dimensions
NUM_IDX_MAPS	Long	Number of indexed dimension mapping entries
NUM_MAPS	Long	Number of dimension mapping entries
NUM_FIELDS	Long	Number of fields
NUM_GEO_FIELDS	Long	Number of geolocation field entries
ORIGIN_CODE	Long	Origin code
PIX_REG_CODE	Long	Pixel registration code

Table 5-1: Fields of the Info Structure (Continued)

Version History

5.3	Introduced

EOS_GD_READATTR

This function reads attributes from the grid.

Syntax

Result = EOS_GD_READATTR(*gridID*, *attrname*, *datbuf*)

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

attrname

Attribute name (string).

datbuf

A named variable that will contain the attribute values.

Keywords

None

Examples

In this example, we read a single precision (32 bit) floating point attribute with the name “ScalarFloat”:

```
status = EOS_GD_READATTR(gridID, "ScalarFloat", f32)
```

Version History

5.2	Introduced

EOS_GD_READFIELD

This function reads data from the grid field. The values within start, stride, and edge arrays refer to the grid field (input) dimensions. The default values for start and stride are 0 and 1 respectively. The default value for edge is (dim - start) / stride where dim refers to the size of the dimension.

Note

Array ordering of variables used or returned by this routine changed in IDL 5.5. Programs written for versions of this routine prior to IDL 5.5 may need to be modified to work correctly with the current version. See "Note on Array Ordering" on page 579 for details.

Syntax

Result = EOS_GD_READFIELD(*gridID*, *fieldname*, *buffer* [, EDGE=*array*]
[, START=*array*] [, STRIDE=*array*])

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

fieldname

Name of field (string) to read.

buffer

A named variable that will contain the data read from the field.

Keywords

EDGE

Array (long) specifying the number of values to read along each dimension.

START

Array (long) specifying the starting location within each dimension.

STRIDE

Set this keyword to an array of integers specifying the number of values to step along each dimension. The default is [1, 1, ...] indicating that every value should be included. Specifying a stride of 0 is equivalent to 1.

Examples

In this example, we read data from the 10th row (0-based) of the Temperature field:

```
start=[10,1]
edge=[1,120]
status = EOS_GD_READFIELD(gridID, "Temperature", row, $
   START = start, EDGE = edge)
```

Version History

5.2	Introduced

EOS_GD_READTILE

This function reads a single tile of data from a field. If the data is to be read tile by tile, this routine is more efficient than EOS_GD_READFIELD. In all other cases, the later routine should be used. EOS_GD_READTILE does not work on non-tiled fields. Note that the coordinates are in terms of tiles, not data elements.

Note

Array ordering of variables used or returned by this routine changed in IDL 5.5. Programs written for versions of this routine prior to IDL 5.5 may need to be modified to work correctly with the current version. See "Note on Array Ordering" on page 579 for details.

Syntax

Result = EOS_GD_READTILE(*gridID*, *fieldname*, *tilecoords*, *buffer*)

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

fieldname

Fieldname (string).

tilecoords

Array (long) of tile coordinates.

buffer

A named variable that will contain the tile.

Keywords

None

Examples

In this example. we read one tile from the Temperature field (see EOS_GD_DEFTILE example) located at the second column of the first row of tiles:

```
tilecoords[0] = 0
tilecoords[1] = 1
status = EOS_GD_READTILE(gridid, "Temperature",$
   tilecoords,buffer)
```

Version History

5.2	Introduced

EOS_GD_REGIONINFO

This function returns information about a subsetted region for a particular field. Because of differences in number type and geolocation mapping, a given region will give different values for the dimensions and size for various fields. The upleftpt and lowrightpt arrays can be used when creating a new grid from the subsetted region.

Note

Array ordering of variables used or returned by this routine changed in IDL 5.5. Programs written for versions of this routine prior to IDL 5.5 may need to be modified to work correctly with the current version. See "Note on Array Ordering" on page 579 for details.

Syntax

Result = EOS_GD_REGIONINFO(*gridID*, *regionID*, *fieldname*, *ntype*, *rank*, *dims*, *size*, *upleftpt*, *lowrightpt*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

regionID

Region or period id (long) returned by EOS_GD_DEFBOXREGION.

fieldname

Field to subset (string).

ntype

A named variable that will contain the HDF data type of field (long).

rank

A named variable that will contain the rank of field (long).

dims

A named variable that will contain the dimensions of subset region (long).

size

A named variable that will contain the size in bytes of subset region (long).

upleftpt

A named variable that will contain the upper left point of subset region (double array).

lowrightpt

A named variable that will contain the lower right point of subset region (double array).

Keywords

None

Examples

In this example, we retrieve information about the region defined in EOS_GD_DEFBOXREGION for the "Temperature" field:

```
status = EOS_GD_REGIONINFO(EOS_GD_id, regionID, $
   "Temperature", ntype,$ rank, dims, size, upleft,$ lowright)
```

Version History

5.2	Introduced

EOS_GD_SETFILLVALUE

This function sets the fill value for the specified field. The fill value is placed in all elements of the field which have not been explicitly defined.

Syntax

Result = EOS_GD_SETFILLVALUE(*gridID*, *fieldname*, *fillvalue*)

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

fieldname

Fieldname (string).

fillvalue

The fill value to be used.

Keywords

None

Examples

In this example, we set a fill value for the “Temperature” field:

```
tempfill = -999.0
status = EOS_GD_SETFILLVALUE(gridID, "Temperature", tempfill)
```

Version History

5.2	Introduced

EOS_GD_SETTILECACHE

This function sets tile cache parameters.

Syntax

Result = EOS_GD_SETTILECACHE(*gridID*, *fieldname*, *maxcache*, *cachecode*)

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

fieldname

Fieldname (string).

maxcache

Maximum number of tiles (long) to cache in memory.

cachecode

Currently must be set to 0 (long).

Keywords

None

Examples

In this example, we set maxcache to 10 tiles. The particular subsetting envisioned for the Spectra field (defined in the EOS_GD_DEFTILE example) would never cross more than 10 tiles along the field's fastest varying dimension, i.e., XDim.

```
status = EOS_GD_SETTILECACHE(gridID, "Spectra", 10, 0)
```

Version History

5.2	Introduced

EOS_GD_TILEINFO

This function returns the tiling code, tiling rank, and tiling dimensions for a given field.

Note

Array ordering of variables used or returned by this routine changed in IDL 5.5. Programs written for versions of this routine prior to IDL 5.5 may need to be modified to work correctly with the current version. See "Note on Array Ordering" on page 579 for details.

Syntax

Result = EOS_GD_TILEINFO(*gridID*, *fieldname*, *tilecode*, *tilerank*, *tiledims*)

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

fieldname

Fieldname (string).

tilecode

A named variable that will contain the tile code (long): 0 No Tile, 1 Tile.

tilerank

A named variable that will contain the number of tile dimensions (long).

tiledims

A named variable that will contain the tile dimensions (long).

Keywords

None

Examples

To retrieve the tiling information about the Pressure field defined in the EOS_GD_DEFTILE section:

```
status = EOS_GD_COMPINFO(gridID, "Pressure", tilecode, $
   tilerank, tiledims)
```

Version History

5.2	Introduced

EOS_GD_WRITEATTR

This function writes/updates attributes in the grid. If the attribute does not exist, it is created. If it does exist, then the value(s) is (are) updated.

Syntax

Result = EOS_GD_WRITEATTR(*gridID*, *attrname*, *datbuf* [, COUNT=*value*]
[, HDF_TYPE=*value*])

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

attrname

Attribute name (string).

datbuf

Attribute values. If HDF_TYPE is specified, the IDL variable is first converted to the type specified by the keyword before being written.

Keywords

COUNT

Number of values to store in attribute (long).

HDF_TYPE

HDF data type of attribute (long). See "IDL and HDF Data Types" on page 317 for valid values.

Examples

In this example. we write a single precision (32 bit) floating point number with the name “ScalarFloat” and the value 3.14:

```
f32 = 3.14
status = EOS_GD_WRITEATTR(gridid, "ScalarFloat", f32)
```

We can update this value by simply calling the function again with the new value:

```
f32 = 3.14159
status = EOS_GD_WRITEATTR(gridid, "ScalarFloat", f32)
```

Version History

5.2	Introduced

EOS_GD_WRITEFIELD

This function writes data to the grid field. The values within start, stride, and edge arrays refer to the grid field (output) dimensions. The input data in the data buffer is read from contiguously. The default values for start and stride are 0 and 1 respectively. The default value for edge is (dim - start) / stride where dim refers to the size of the dimension. Note that the data buffer for a compressed field must be the size of the entire field as incremental writes are not supported by the underlying HDF routines.

Note

Array ordering of variables used or returned by this routine changed in IDL 5.5. Programs written for versions of this routine prior to IDL 5.5 may need to be modified to work correctly with the current version. See "Note on Array Ordering" on page 579 for details.

Syntax

Result = EOS_GD_WRITEFIELD(*gridID*, *fieldname*, *data* [, EDGE=*array*] [, START=*array*] [, STRIDE=*array*])

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

fieldname

Name of field (string) to write.

data

Values (long) to be written to the field.

Keywords

EDGE

Array (long) specifying the number of values to write along each dimension.

START

Array (long) specifying the starting location within each dimension (0-based).

STRIDE

Set this keyword to an array of integers specifying the number of values to step along each dimension. The default is [1, 1, ...] indicating that every value should be included. Specifying a stride of 0 is equivalent to 1.

Examples

In this example, we write data to the Temperature field:

```
; Define elements of temperature array:
temperature = indegen (200, 120)
status = EOS_GD_WRITEFIELD(gridID, "Temperature", temperature)

; Update Row 10 (0-based) in this field:
start=[0,10], edge=[2000,1]

; Define elements of newrow array:
status = EOS_GD_WRITEFIELD(gridID, "Temperature", $
   START=start, EDGE=edge, newrow)
```

Version History

5.2	Introduced

EOS_GD_WRITEFIELDMETA

This function writes the field metadata for a grid field not defined by the Grid API.

Note

Array ordering of variables used or returned by this routine changed in IDL 5.5. Programs written for versions of this routine prior to IDL 5.5 may need to be modified to work correctly with the current version. See "Note on Array Ordering" on page 579 for details.

Syntax

Result = EOS_GD_WRITEFIELDMETA(*gridID*, *fieldname*, *dimlist*, *numbertype*)

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

fieldname

Name of field (string) that metadata information is to be written.

dimlist

Dimension list of field (long).

numbertype

Number type of data in field (long).

Keywords

None

Examples

```
status = EOS_GD_writefieldmeta(gridID, "ExternField", $
   "Ydim,Xdim", 5)
```

Version History

5.2	Introduced

EOS_GD_WRITETILE

This function writes a single tile of data to a field. If the data to be written to a field can be arranged tile by tile, this routine is more efficient than EOS_GD_WRITEFIELD. In all other cases, the EOS_GD_WRITEFIELD routine should be used. EOS_GD_WRITETILE does not work on non-tiled fields. Note that the are coordinates in terms of tiles, not data elements.

Note

Array ordering of variables used or returned by this routine changed in IDL 5.5. Programs written for versions of this routine prior to IDL 5.5 may need to be modified to work correctly with the current version. See "Note on Array Ordering" on page 579 for details.

Syntax

Result = EOS_GD_WRITETILE(*gridID*, *fieldname*, *tilecoords*, *data*)

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise.

Arguments

gridID

Grid id (long) returned by EOS_GD_CREATE or EOS_GD_ATTACH.

fieldname

Fieldname (string).

tilecoords

Array of tile coordinates (long).

data

Data to be written to tile.

Keywords

None

Examples

In this example, we write one tile to the Temperature field (see the EOS_GD_DEFTILE example) at the second column of the first row of tiles:

```
tilecoords[0] = 0
tilecoords[1] = 1
status=EOS_GD_WRITETILE(gridID, "Temperature", tilecoords, data)
```

Version History

5.2	Introduced

EOS_PT_ATTACH

This function attaches to the point using the pointname parameter as the identifier.

Syntax

Result = EOS_PT_ATTACH(*fid*, *pointname*)

Return Value

Returns the point handle (pointID) if successful and FAIL (–1) otherwise.Typical reasons for failure are an improper point file id or point name.

Arguments

fid

Point file id (long) returned by EOS_PT_OPEN.

pointname

Name of point (string) to be attached.

Keywords

None

Examples

In this example, we attach to the previously created point, “ExamplePoint”, within the HDF file, PointFile.hdf, referred to by the handle, fid:

```
pointID = EOS_PT_ATTACH(fid, "ExamplePoint")
```

The point can then be referenced by subsequent routines using the handle, pointID.

Version History

5.2	Introduced

See Also

EOS_PT_DETACH

EOS_PT_ATTRINFO

This function returns number type and number of elements (count) of a point attribute.

Syntax

Result = EOS_PT_ATTRINFO(*pointID*, *attrname*, *numbertype*, *count*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long) returned by EOS_PT_CREATE or EOS_PT_ATTACH.

attrname

Attribute name (string).

numbertype

A named variable that will contain the HDF type of the attribute value (long).

count

A named variable that will contain the number of total bytes in attribute (long).

Keywords

None

Examples

In this example, we return information about the ScalarFloat attribute:

```
status = EOS_PT_ATTRINFO(pointID, "ScalarFloat", nt, count)
```

Version History

5.2	Introduced

EOS_PT_BCKLINKINFO

This function returns the linkfield to the previous level.

Syntax

Result = EOS_PT_BCKLINKINFO(*pointID*, *level*, *linkfield*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long) returned by EOS_PT_CREATE or EOS_PT_ATTACH.

level

Point level (0-based long).

linkfield

A named variable that will contain the link field (string).

Keywords

None

Examples

In this example, we return the linkfield connecting the Observations level to the previous Desc-Loc level. (These levels are defined in the EOS_PT_DEFLEVEL routine.)

```
status = EOS_PT_BCKLINKINFO(pointID2, 1, linkfield)
```

Version History

5.2	Introduced

EOS_PT_CLOSE

This function closes the HDF point file.

Syntax

Result = EOS_PT_CLOSE(*fid*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

fid

Point file id (long) returned by EOS_PT_OPEN.

Keywords

None

Examples

```
status = EOS_PT_CLOSE(fid)
```

Version History

5.2	Introduced

EOS_PT_CREATE

This function creates a new point structure. The point is created as a Vgroup within the HDF file with the name *pointname* and class POINT.

Syntax

Result = EOS_PT_CREATE(*fid*, *pointname*)

Return Value

Returns the point handle (pointID) if successful and FAIL (–1) otherwise.

Arguments

fid

Point file id (long) returned by EOS_PT_OPEN.

pointname

Name of point (string) to be created.

Keywords

None

Examples

In this example, we create a new point structure, ExamplePoint, in the previously created file, PointFile.hdf:

```
pointID = EOS_PT_CREATE(fid, "ExamplePoint")
```

The point structure is then referenced by subsequent routines using the handle, pointID.

Version History

5.2	Introduced

EOS_PT_DEFBOXREGION

This function defines an area of interest for a point. It returns a point region ID which is used by the EOS_PT_EXTRACTREGION routine to read the fields from a level for those records within the area of interest.The point structure must have a level with both a Longitude and Latitude (or Colatitude) field defined.

Syntax

Result = EOS_PT_DEFBOXREGION(*pointID*, *cornerlon*, *cornerlat*)

Return Value

Returns the point regionID if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long) returned by EOS_PT_CREATE or EOS_PT_ATTACH.

cornerlon

Longitude (double) in decimal degrees of box corners (2 element, 1-D array).

cornerlat

Latitude (double) in decimal degrees of box corners (2 element, 1-D array).

Keywords

None

Examples

In this example, we define an area of interest with (opposite) corners at –145 degrees longitude, –15 degrees latitude and –135 degrees longitude, –8 degrees latitude:

```
cornerlon = dblarr (2)
cornerlat = dblarr (2)
cornerlon[0] = -145.
cornerlat[0] = -15.
cornerlon[1] = -135.
```

```
cornerlat[1] = -8.
regionID = EOS_PT_DEFBOXREGION(pointID, cornerlon, cornerlat)
```

Version History

5.2	Introduced

EOS_PT_DEFLEVEL

This function defines a level within the point. A simple point consists of a single level. A point where there is common data for a number of records can be more efficiently stored with multiple levels. The order in which the levels are defined determines the (0-based) level index.

Syntax

Result = EOS_PT_DEFLEVEL(*pointID*, *levelname*, *fieldlist*, *fieldtype*, *fieldorder*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long) returned by EOS_PT_CREATE or EOS_PT_ATTACH.

levelname

Name of level (string) to be defined.

fieldlist

List of fields (string) in level.

fieldtype

Array (long) containing HDF data type of each field within level.

fieldorder

Array (long) containing order of each field within level.

Note

An order of 0 is considered the same as an order of 1.

Keywords

None

Examples

Example 1 — Simple Point:

In this example, we define a simple single level point, with levelname, Sensor. The levelname should not contain any slashes ("/"). It consists of six fields, ID, Time, Longitude, Latitude, Temperature, and Mode defined in the field list. The fieldtype and fieldorder parameters are arrays consisting of the HDF number type codes and field orders, respectively. The Temperature is an array field of dimension 4 and the Mode field a character string of size 4. All other fields are scalars. Note that the order for numerical scalar variables can be either 0 or 1.

```
fieldtype = [22, 22, 5, 5, 5, 4]
fieldorder = [0,0,0,0,4,4]
fldlist = "ID,Time,Longitude,Latitude,Temperature,Mode"
status = EOS_PT_DEFLEVEL(pointID, "Sensor", fldlist, fieldtype,$
fieldorder)
```

Example 2 — Multi-Level Point:

In this example, we define a two-level point that describes data from a network of fixed buoys. The first level contains information about each buoy and includes the name (label) of the buoy, its (fixed) longitude and latitude, its deployment date, and an ID that is used to *link* it to the following level. (The link field is defined in the EOS_PT_DEFLINKAGE routine described later.) The entries within the ID field must be unique. The second level contains the actual measurements from the buoys (rainfall and temperature values) plus the observation time and the ID which relates a given measurement to a particular buoy entry in the previous level. There can be many records in this level with the same ID since there can be multiple measurements from a single buoy. It is advantageous, although not mandatory, to store all records for a particular buoy (ID) contiguously.

Level 0

```
fieldtype0 = [4, 6, 6, 5, 4]
fieldorder0 = [8,0,0,0,1]
fldlist0 =       "Label,Longitude,Latitude,DeployDate,ID"
status = EOS_PT_deflevel(pointID2, "Desc-Loc", $
   fldlist0, fieldtype0, fieldorder0)
```

Level 1

```
fieldtype1 = [6, 5, 5, 4]
fieldorder1 = [0,0,0,1]
fldlist1 = "Time,Rainfall,Temperature,ID"
status = EOS_PT_DEFLEVEL(pointID2, "Observations", $
   fldlist1, fieldtype1, fieldorder1)
```

Version History

5.2	Introduced

EOS_PT_DEFLINKAGE

This function defines the linkfield between two levels. This field must be defined in both levels.

Syntax

Result = EOS_PT_DEFLINKAGE(*pointID*, *parent*, *child*, *linkfield*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long) returned by EOS_PT_CREATE or EOS_PT_ATTACH.

parent

Name (string) of parent level.

child

Name (string) of child level.

linkfield

Name (string) of common linkfield.

Keywords

None

Examples

In this example, we define the ID field as the link between the two levels defined previously in the EOS_PT_DEFLEVEL function:

```
status = EOS_PT_DEFLINKAGE(pointID2, "Desc-Loc", $
   "Observations", "ID")
```

Version History

5.2	Introduced

EOS_PT_DEFTIMEPERIOD

This function defines a time period for a point. It returns a point period ID which is used by the EOS_PT_EXTRACTPERIOD function to read the fields from a level for those records within the time period. The point structure must have a level with the Time field defined.

Syntax

Result = EOS_PT_DEFTIMEPERIOD(*pointID*, *starttime*, *stoptime*)

Return Value

Returns the point periodID if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long) returned by EOS_PT_CREATE or EOS_PT_ATTACH.

starttime

Start time (double) of period.

stoptime

Stop time (double) of period.

Keywords

None

Examples

In this example, we define a time period with a start time of 35208757.6 and a stop time of 35984639.2:

```
starttime = 35208757.6d
stoptime = 35984639.2d
periodID = EOS_PT_DEFTIMEPERIOD(pointID, starttime, stoptime)
```

Version History

5.2	Introduced

EOS_PT_DEFVRTREGION

This function allows the user to select those records within a point whose field values are within a given range. (For the current version of this routine, the field must have one of the following HDF data types: 22, 24, 5, 6) This function may be called after EOS_PT_DEFBOXREGION or EOS_PT_DEFTIMEPERIOD to provide both geographic or time and vertical subsetting. In this case the user provides the id from the previous subset call. (This same id is then returned by the function.) This routine may also be called stand-alone by setting the input id to (–1).

This function may be called up to eight times with the same region ID. In this way a region can be subsetted along a number of dimensions.

The EOS_PT_REGIONINFO and EOS_PT_EXTRACTREGION functions work as before, however, because there is no mapping performed between geolocation dimensions and data dimensions for the field to be subsetted, (the field specified in the call to EOS_PT_REGIONINFO and EOS_PT_EXTRACTREGION) must contain the dimension used explicitly in the call to EOS_PT_DEFVRTREGION (case 1) or the dimension of the one-dimensional field (case 2).

Syntax

Result = EOS_PT_DEFVRTREGION(*pointID*, *regionID*, *vertObj*, *range*)

Return Value

Returns the point region ID if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long) returned by EOS_PT_CREATE or EOS_PT_ATTACH.

regionID

Region or period id (long) returned from a previous subset call.

vertObj

String name of a dimension or field by which to subset.

range

Minimum and maximum range for the subset (double, 2 element, 1-D array).

Keywords

None

Examples

Suppose we wish to find those records within a point whose Rainfall values fall between 1 and 2. We wish to search all the records within the point, so we set the input region ID to (–1):

```
range = [1.,2.]
regionID = EOS_PT_DEFVRTREGION(pointID, -1, "Rainfall", range)

; Now we subset further using the Temperature field:
range = [22.,24.]
regionID = EOS_PT_DEFVRTREGION(pointID, regionID, $
   "Temperature", range)
```

The subsetted region referred to by regionID will now contain those records whose Rainfall field are between 1 and 2 and whose Temperature field are between 22 and 24.

Version History

5.2	Introduced

EOS_PT_DETACH

This function detaches from a point data set. This function should be run before exiting from the point file for every point opened by EOS_PT_CREATE or EOS_PT_ATTACH.

Syntax

Result = EOS_PT_DETACH(*pointID*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long) returned by EOS_PT_CREATE or EOS_PT_ATTACH

Keywords

None

Examples

```
status = EOS_PT_DETACH(pointID)
```

Version History

5.2	Introduced

See Also

EOS_PT_ATTACH

EOS_PT_EXTRACTPERIOD

This function reads data from the designated level fields into the data buffer from the subsetted time period.

Syntax

Result = EOS_PT_EXTRACTPERIOD(*pointID*, *periodID*, *level*, *fieldlist*, *buffer*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long).

periodID

Period id (long) returned by EOS_PT_DEFTIMEPERIOD.

level

Point level (0-based long).

fieldlist

List of fields (string) to extract.

buffer

A named variable that will contain the data buffer. This buffer is in packed format. Use HDF_UNPACKDATA to convert it into variables.

Keywords

None

Examples

In this example, we read data within the subsetted time period defined by EOS_PT_DEFTIMEPERIOD from the Time field:

```
periodID = EOS_PT_DEFTIMEPERIOD(pointID, 35208757.6d, $
   35984639.2d)
IF (periodID NE -1) THEN BEGIN
   status = EOS_PT_EXTRACTPERIOD(pointID, periodID, 1, $
      "Time", buffer)
   HDF_UNPACKDATA, buffer, dataTime, HDF_TYPE=[6]
ENDIF
```

Version History

5.2	Introduced

EOS_PT_EXTRACTREGION

This function reads data from the designated level fields into the data buffer from the subsetted area of interest.

Syntax

Result = EOS_PT_EXTRACTREGION(*pointID*, *regionID*, *level*, *fieldlist*, *buffer*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long).

regionID

Period id (long) returned by EOS_PT_DEFBOXREGION.

level

Point level (0-based long).

fieldlist

List of fields (string) to extract.

buffer

A named variable that will contain the data buffer.

Keywords

None

Examples

In this example, we read data within the subsetted area of interest defined by EOS_PT_DEFBOXREGION from the Longitude and Latitude fields:

```
regionID = EOS_PT_DEFBOXREGION(pointID, [-145.,-135.], [-15.,-8.])
IF (regionID NE -1) THEN BEGIN
   status = EOS_PT_EXTRACTREGION(pointID, regionID, 0, $
      "Longitude,Latitude", buffer)
   HDF_UNPACKDATA, buffer, dataLong,dataLat,HDF_TYPE=[6,6]
ENDIF
```

Version History

5.2	Introduced

EOS_PT_FWDLINKINFO

This function returns the linkfield to the given level.

Syntax

Result = EOS_PT_FWDLINKINFO(*pointID*, *level*, *linkfield*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long) returned by EOS_PT_CREATE or EOS_PT_ATTACH.

level

Point level (0-based long).

linkfield

A named variable that will contain the link field (string).

Keywords

None

Examples

In this example, we return the linkfield connecting the Desc-Loc level to the following Observations level. (These levels are defined in the EOS_PT_DEFLEVEL function.):

```
status = EOS_PT_FWDLINKINFO(pointID2, 1, linkfield)
```

Version History

5.2	Introduced

EOS_PT_GETLEVELNAME

This function returns the name of a level given the level number (0-based).

Syntax

Result = EOS_PT_GETLEVELNAME(*pointID*, *level*, *levelname*
[, LENGTH=*variable*])

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long) returned by EOS_PT_CREATE or EOS_PT_ATTACH.

level

Point level (0-based long).

levelname

A named variable that will contain the level name (string).

Keywords

LENGTH

Set this keyword to a named variable that will contain the string length of the level name.

Examples

In this example, we return the level name of the 0th level of the second point defined in the EOS_PT_DEFLEVEL section:

```
status = EOS_PT_GETLEVELNAME(pointID2, 0, levelname)
```

Version History

5.2	Introduced

EOS_PT_GETRECNUMS

This function returns the record numbers in one level that are connected to a given set of records in a different level. The two levels need not be adjacent. The records in one level are related to those in another through the link field. These in turn are related to the next. In this way, each record in any level is related to others in all the levels of the point structure.

Syntax

Result = EOS_PT_GETRECNUMS(*pointID*, *inlevel*, *outlevel*, *inNrec*, *inRecs*, *outNrec*, *outRecs*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long) returned by EOS_PT_CREATE or EOS_PT_ATTACH.

inlevel

Level number (long) of input records (0-based).

outlevel

Level number (long) of output records (0-based).

inNrec

Number of records (long) in the inRecs array.

inRecs

Array (long) containing the input record numbers.

outNrec

A named variable that will contain the number of records (long) in the outRecs array.

outRecs

A named variable that will contain the array (long) of output record numbers.

Keywords

None

Examples

In this example, we get the record numbers in the second level that are related to the first record in the first level:

```
nrec = 1
recs[0] = 0
inLevel = 0
outLevel = 1
status = EOS_PT_GETRECNUMS(pointID2, inLevel, outLevel, $
   nrec, recs, outNrec, outRecs)
```

Version History

5.2	Introduced

EOS_PT_INQATTRS

This function retrieves information about the attributes defined in a point structure. The attribute list is returned as a string with each attribute name separated by a comma.

Note
See STRSPLIT to separate the attribute list.

Syntax

Result = EOS_PT_INQATTRS(*pointID*, *attrlist* [, LENGTH=*variable*])

Return Value

Number of attributes found or (–1) if failure.

Arguments

pointID

Point id (long).

attrlist

A named variable that will contain the attribute list (string) entries separated by commas.

Keywords

LENGTH

Set this keyword to a named variable that will contain the length of the attribute list, as a long integer.

Examples

```
nattr = EOS_PT_INQATTRS(pointID, attrlist)
```

Version History

5.2	Introduced

EOS_PT_INQPOINT

This function retrieves the number and names of points defined in an HDF-EOS file. The point list is returned as a string with each point name separated by a comma.

Note

See STRSPLIT to separate the attribute list.

Syntax

Result = EOS_PT_INQPOINT*(filename*, *pointlist* [, LENGTH=*variable*])

Return Value

Returns number of points found or (–1) if failure.

Arguments

filename

HDF-EOS filename (string).

pointlist

A named variable that will contain the point list (string) entries separated by commas.

Keywords

LENGTH

Set this keyword to a named variable that will contain the length of the point list as a long integer.

Examples

In this example, we retrieve information about the points defined in an HDF-EOS file, HDFEOS.hdf:

```
npoint = EOS_PT_INQPOINT("HDFEOS.hdf", pointlist)
```

Version History

5.2	Introduced

EOS_PT_LEVELINDX

This function returns the level index for a given level specified by name.

Syntax

Result = EOS_PT_LEVELINDX(*pointID*, *levelname*)

Return Value

Returns the level index if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long) returned by EOS_PT_CREATE or EOS_PT_ATTACH.

levelname

Level Name (string).

Keywords

None

Examples

In this example, we return the level index of the Observations level in the multilevel point structure defined in EOS_PT_DEFLEVEL:

```
levindx = EOS_PT_LEVELINDEX(pointID2, "Observations")
```

Version History

5.2	Introduced

EOS_PT_LEVELINFO

This function returns information about the fields in a given level. Typical reasons for failure are an improper point id or level number.

Syntax

Result = EOS_PT_LEVELINFO(*pointID*, *level*, *fieldlist*, *fldtype*, *fldorder*)

Return Value

Returns number of fields if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long) returned by EOS_PT_CREATE or EOS_PT_ATTACH.

level

Point level (0-based long).

fieldlist

A named variable that will contain field names (string) in level.

fldtype

A named variable that will contain the number HDF data type (long) of each field.

fldorder

A named variable that will contain the order (long) of each field.

Keywords

None

Examples

In this example, we return information about the Desc-Loc (1st) level defined previously:

```
nflds = EOS_PT_LEVELINFO(pointID2, 0, fldlist, fldtype, fldorder)
```

The last variable is useful only when information on an entire point is requested.

Version History

5.2	Introduced

EOS_PT_NFIELDS

This function returns the number of fields in a level.

Syntax

Result = EOS_PT_NFIELDS(*pointID*, *level* [, LENGTH=*bytes*])

Return Value

Returns number of fields if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long) returned by EOS_PT_CREATE or EOS_PT_ATTACH.

level

Level number (0-based long).

Keywords

LENGTH

Size (long) in bytes of fieldlist for level.

Examples

In this example, we retrieve the number of fields in the 2nd point defined previously:

```
nflds=EOS_PT_NFIELDS(pointID2,0)
```

Version History

5.2	Introduced

EOS_PT_NLEVELS

This function returns the number of levels in a point.

Syntax

Result = EOS_PT_NLEVELS(*pointID*)

Return Value

Returns number of levels if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long) returned by EOS_PT_CREATE or EOS_PT_ATTACH.

Keywords

None

Examples

In this example, we retrieve the number of levels in the 2nd point defined previously:

```
nlevels = EOS_PT_NLEVELS(pointID2)
```

Version History

5.2	Introduced

EOS_PT_NRECS

This function returns the number of records in a given level.

Syntax

Result = EOS_PT_NRECS(*pointID*, *level*)

Return Value

Returns number of records in a given level if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long) returned by EOS_PT_CREATE or EOS_PT_ATTACH.

level

Level number (0-based long).

Keywords

None

Examples

In this example, we retrieve the number of records in the first level of the 2nd point defined previously:

```
nrecs = EOS_PT_NRECS(pointID2, 0)
```

Version History

5.2	Introduced

EOS_PT_OPEN

This function creates a new file or opens an existing one.

Syntax

Result = EOS_PT_OPEN(*fieldname* [, /CREATE] [, /RDWR | , /READ])

Return Value

Returns the point file id handle (fid) if successful and FAIL (–1) otherwise.

Arguments

fieldname

Complete path and filename (string) for the file to be opened.

Keywords

CREATE

If file exists, delete it, then open a new file for read/write.

RDWR

Open for read/write. If file does not exist, create it.

READ

Open for read only. If file does not exist then error.

Examples

In this example, we create a new point file named, PointFile.hdf. It returns the file handle, fid.

```
fid = EOS_PT_OPEN("PointFile.hdf", /CREATE)
```

Version History

5.2	Introduced

See Also

EOS_PT_CLOSE

EOS_PT_PERIODINFO

This function returns information about a subsetted time period for a particular fieldlist.

Syntax

Result = EOS_PT_PERIODINFO(*pointID*, *periodID*, *level*, *fieldlist*, *size*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long).

periodID

Period id (long) returned by EOS_PT_DEFTIMEPERIOD.

level

Point level (0-based long).

fieldlist

List of fields (string) to extract.

size

A named variable that will contain the size in bytes (long) of subset period.

Keywords

None

Examples

In this example, we get the size of the subsetted time period defined in EOS_PT_DEFTIMEPERIOD for the Time field:

```
status = EOS_PT_PERIODINTO(pointID, periodID, 0, "Time", size)
```

Version History

5.2	Introduced

EOS_PT_PERIODRECS

This function returns the record numbers within a subsetted time period for a particular level.

Syntax

Result = EOS_PT_PERIODRECS(*pointID*, *periodID*, *level*, *nrec*, *recs*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long).

periodID

Period id (long) returned by EOS_PT_DEFTIMEPERIOD.

level

Point level (0-based long).

nrec

A named variable that will contain the number of records (long) within time period in level.

recs

A named variable that will contain the record numbers (long) of subsetted records in level.

Keywords

None

Examples

In this example, we get the number of records and record numbers within the subsetted area of interest defined in EOS_PT_DEFTIMEPERIOD for the 0th level:

```
status = EOS_PT_PERIODRECS(pointID, periodID, 0, nrec, recs)
```

Version History

5.2	Introduced

EOS_PT_QUERY

The EOS_PT_QUERY function returns information about a specified point.

Syntax

Result = EOS_PT_QUERY(*Filename*, *PointName*, [*Info*])

Return Value

This function returns an integer value of 1 if the file is an HDF file with EOS POINT extensions, and 0 otherwise.

Arguments

Filename

A string containing the name of the file to query.

PointName

A string containing the name of the point to query.

Info

Returns an anonymous structure containing information about the specified point. The returned structure contains the following fields:

Field	IDL Data Type	Description
ATTRIBUTES	String array	Array of attribute names
NUM_ATTRIBUTES	Long	Number of attributes
NUM_LEVELS	Long	Number of levels

Table 5-2: Fields of the Info Structure

Keywords

None

Version History

5.3	Introduced

EOS_PT_READATTR

This function reads attributes.

Syntax

Result = EOS_PT_READATTR(*pointID*, *attrname*, *datbuf*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long) returned by EOS_PT_CREATE or EOS_PT_ATTACH.

attrname

Attribute name (string).

datbuf

A named variable that will contain the buffer allocated to hold attribute values.

Keywords

None

Examples

In this example, we read a single precision (32 bit) floating point attribute with the name “ScalarFloat”:

```
status = EOS_PT_READATTR(pointID, "ScalarFloat", f32)
```

Version History

5.2	Introduced

EOS_PT_READLEVEL

This function reads data from the specified fields and records of a single level in a point.

Syntax

Result = EOS_PT_READLEVEL(*pointID*, *level*, *fieldlist*, *nrec*, *recs*, *buffer*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long) returned by EOS_PT_CREATE or EOS_PT_ATTACH.

level

Level to read (0-based long).

fieldlist

List of fields (string) to read.

nrec

Number of records (long) to read.

recs

Record number of records to read (0-based long).

buffer

A named variable that will contain the buffer to store data. This buffer is in packed format. Use HDF_UNPACKDATA to convert it into IDL variables.

Keywords

None

Examples

In this example, we read records 0, 2, and 3 from the Temperature and Mode fields in the first level of the point referred to by point ID, pointID. Temperature is a 32-bit float field and Mode is a 4 character field (HDF types 5 and 4 respectively):

```
recs = [ 0, 2, 3 ]
status = EOS_PT_READLEVEL( pointID, 0, "Temperature,Mode", $
   3, recs, buffer)
IF (status EQ 0) THEN BEGIN
   HDF_UNPACKDATA, buffer, dataTemperature, dataMode, $
      HDF_TYPE=[5,4], HDF_ORDER = [4,4]
ENDIF
```

Version History

5.2	Introduced

EOS_PT_REGIONINFO

This function returns information about a subsetted area of interest for a particular fieldlist.

Syntax

Result = EOS_PT_REGIONINFO(*pointID*, *regionID*, *level*, *fieldlist*, *size*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long).

regionID

Region id (long) returned by EOS_PT_DEFBOXREGION.

level

Point level (0-based long).

fieldlist

List of fields (sting) to extract.

size

A named variable that will contain the size in bytes (long) of subset period.

Keywords

None

Examples

In this example, we get the size of the subsetted area of interest defined in EOS_PT_DEFBOXREGION from the Longitude and Latitude fields:

```
status = EOS_PT_REGIONINFO(pointID, regionID, 0, "Longitude, $
   Latitude",size)
```

Version History

5.2	Introduced

EOS_PT_REGIONRECS

This function returns the record numbers within a subsetted geographic region for a particular level.

Syntax

Result = EOS_PT_REGIONRECS(*pointID*, *regionID*, *level*, *nrec*, *recs*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long).

regionID

Region id (long) returned by EOS_PT_DEFBOXREGION.

level

Point level (0-based long).

nrec

A named variable that will contain the number of records (long) within geographic region in level.

recs

A named variable that will contain the record numbers (long) of subsetted records in level.

Keywords

None

Examples

In this example, we get the number of records and record numbers within the subsetted area of interest defined in EOS_PT_DEFBOXREGION for the 0th level:

```
status = EOS_PT_REGIONRECS(pointID, regionID, 0, nrec, recs)
```

Version History

5.2	Introduced

EOS_PT_SIZEOF

This function returns information about specified fields in a point regardless of level.

Syntax

Result = EOS_PT_SIZEOF(*pointID*, *fieldlist*, *fldlevel*)

Return Value

Returns size in bytes of specified fields and FAIL (–1) otherwise.

Arguments

pointID

Point id (long) returned by EOS_PT_CREATE or EOS_PT_ATTACH.

fieldlist

Field names (string).

fldlevel

A named variable that will contain the level number (long) of each field.

Keywords

None

Examples

In this example, we return the size in bytes of the Label and Rainfall fields in the 2nd point defined in the EOS_PT_DEFLEVEL function:

```
size = EOS_PT_SIZEOF(pointID2, "Label,Rainfall", fldlevel)
```

Version History

5.2	Introduced

EOS_PT_UPDATELEVEL

This function updates the specified fields and records of a single level.

Syntax

Result = EOS_PT_UPDATELEVEL(*pointID*, *level*, *field*, *list*, *nrec*, *recs*, *data*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long) returned by EOS_PT_CREATE or EOS_PT_ATTACH.

level

Level to update (0-based long).

fieldlist

List of fields (string) to update.

nrec

Number of records (long) to update.

recs

Record number of records to update (0-based long).

data

Values to be written to the fields. Data values are not converted to the internal HDF type automatically. Use HDF_PACKDATA if conversion is necessary or the data fields specify multiple types.

Keywords

None

Examples

In this example, we update records 0, 2, and 3 in the Temperature and Mode fields in the second level in the point referred to by the point ID pointID. Temperature is a 4 value 32-bit float field and Mode is a 4 character field (HDF types 5 and 4 respectively):

```
recs = [ 0, 2, 3]
dataTemperature = [ [20, 21, 22, 23], [30, 31, 32, 33], $
   [40, 41, 42, 43]]]
dataMode = ['P', 'I', 'A']
buffer = HDF_PACKDATA(dataTemperature, dataMode, $
   HDF_TYPE = [5, 4], HDF_ORDER = [4, 4])
status = EOS_PT_UPDATELEVEL( pointID, 1, "Temperature,Mode", $
   3, recs, buffer)
```

Version History

5.2	Introduced

EOS_PT_WRITEATTR

This function writes/updates an attribute in a point. If the attribute does not exist, it is created. If it does exist, then the value(s) is (are) updated.

Syntax

Result = EOS_PT_WRITEATTR(*pointID*, *attrname*, *datbuf* [, COUNT=*value*] [, HDF_TYPE=*value*])

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long) returned by EOS_PT_CREATE or EOS_PT_ATTACH.

attrname

Attribute name (string).

datbuf

Attribute values.

Keywords

COUNT

Number of values (long) to store in attribute.

HDF_TYPE

Number type (long) of attribute. See “IDL and HDF Data Types” on page 317 for valid values.

Examples

In this example, we write a single precision (32 bit) floating point number with the name “ScalarFloat” and the value 3.14:

```
f32 = 3.14f
status = EOS_PT_WRITEATTR(pointid, "ScalarFloat", f32)
```

We can update this value by simply calling the function again with the new value:

```
f32 = 3.14159
status = EOS_PT_WRITEATTR(pointid, "ScalarFloat", f32)
```

Version History

5.2	Introduced

EOS_PT_WRITELEVEL

This function writes (appends) full records to a level. The data in each record must be packed. Refer to the section on Vdatas in the HDF documentation. The input data buffer must be sufficient to fill the number of records designated.

Syntax

Result = EOS_PT_WRITELEVEL(*pointID*, *level*, *nrec*, *data*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

pointID

Point id (long) returned by EOS_PT_CREATE or EOS_PT_ATTACH.

level

Level to write (0-based long).

nrec

Number of records (long) to write.

data

Values to be written to the field. Data values are not converted to the internal HDF type automatically. Use HDF_PACKDATA if conversion is necessary or the data fields specify multiple types.

Examples

In this example, we write 5 records to the first level in the point referred to by the point id, pointID1:

```
status = EOS_PT_WRITELEVEL(pointID1, 0, 5, datbuf)
```

Version History

5.2	Introduced

EOS_QUERY

The EOS_QUERY function returns information about the makeup of an HDF-EOS file.

Syntax

Result = EOS_QUERY(*Filename*, [*Info*])

Return Value

This function returns integer value of 1 if the file is an HDF file with EOS extensions, and 0 otherwise.

Arguments

Filename

A scalar string containing the name of the file to query.

Info

Returns an anonymous structure containing information about the contents of the file. The returned structure contains the following fields:

Field	IDL Data Type	Description
GRID_NAMES	String array	Names of grids
NUM_GRIDS	Long	Number of grids in file
NUM_POINTS	Long	Number of points in file
NUM_SWATHS	Long	Number of swaths in file
POINT_NAMES	String array	Names of points
SWATH_NAMES	String array	Names of swaths

Table 5-3: Fields of the Info Structure

Version History

5.3	Introduced

EOS_SW_ATTACH

This function attaches to the swath using the swathname parameter as the identifier.

Syntax

Result = EOS_SW_ATTACH(*fid*, *swathname*)

Return Value

Returns the swath handle (swathID) if successful and FAIL (–1) otherwise.

Arguments

fid

Swath file id (long) returned by EOS_SW_OPEN.

swathname

Name of swath (string) to be attached.

Keywords

None

Examples

In this example, we attach to the previously created swath, “ExampleSwath”, within the HDF file, SwathFile.hdf, referred to by the handle, fid:

```
swathID = EOS_SW_ATTACH(fid, "ExampleSwath")
```

The swath can then be referenced by subsequent routines using the handle, swathID.

Version History

5.2	Introduced

See Also

EOS_SW_DETACH

EOS_SW_ATTRINFO

This function returns the number type and number of elements (count) of a swath attribute.

Syntax

Result = EOS_SW_ATTRINFO(*swathID*, *attrname*, *numbertype*, *count*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

attrname

Attribute name (string).

numbertype

A named variable that will contain the HDF data type (long) of attribute.

count

A named variable that will contain the number of total bytes (long) in attribute.

Keywords

None

Examples

In this example, we return information about the ScalarFloat attribute:

```
status = EOS_SW_ATTRINFO(pointID, "ScalarFloat", nt, count)
```

Version History

5.2	Introduced

EOS_SW_CLOSE

This function closes the HDF swath file.

Syntax

Result = EOS_SW_CLOSE(*fid*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

fid

Swath file id (long) returned by EOS_SW_OPEN.

Keywords

None

Examples

```
status = EOS_SW_CLOSE(fid)
```

Version History

5.2	Introduced

EOS_SW_COMPINFO

This function returns the compression code and compression parameters for a given field.

Syntax

Result = EOS_SW_COMPINFO(*swathID*, *fieldname*, *compcode*, *compparm*)

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

fieldname

Fieldname (string).

compcode

A named variable that will contain the HDF compression code (long).

compparm

A named variable that will contain the compression parameters (long).

Keywords

None

Examples

To retrieve the compression information about the Opacity field defined in the EOS_SW_DEFCOMP section:

```
status = EOS_SW_COMPINFO(swathID, "Opacity", compcode, compparm)
```

Version History

5.2	Introduced

EOS_SW_CREATE

This function creates a swath within the file. The swath is created as a Vgroup within the HDF file with the name *swathname* and class EOS_SWATH.

Syntax

Result = EOS_SW_CREATE(*fid*, *swathname*)

Return Value

Returns the swath handle (swathID) if successful and FAIL (–1) otherwise.

Arguments

fid

Swath file id (long) returned by EOS_SW_OPEN.

swathname

Name of swath (string) to be created.

Keywords

None

Examples

In this example, we create a new swath structure, “ExampleSwath”, in the previously created file, SwathFile.hdf.

```
swathID = EOS_SW_CREATE(fid, "ExampleSwath")
```

The swath structure is referenced by subsequent routines using the handle, swathID.

Version History

5.2	Introduced

EOS_SW_DEFBOXREGION

This function defines a longitude-latitude box region for a swath. It returns a swath region ID that is used by the EOS_SW_EXTRACTREGION function to read all the entries of a data field within the region. A cross track is within a region if its midpoint is within the longitude-latitude box (0), or either of its endpoints is within the longitude-latitude box (1), or any point of the cross track is within the longitude-latitude box (2), depending on the inclusion mode designated by the user. All elements within an included cross track are considered to be within the region even though a particular element of the cross track might be outside the region. The swath structure must have both Longitude and Latitude (or Colatitude) fields defined.

Syntax

Result = EOS_SW_DEFBOXREGION(*swathID*, *cornerlon*, *cornerlat*, *mode*)

Return Value

Returns the swath region ID if successful and FAIL (–1) otherwise.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

cornerlon

Longitude in decimal degrees (double) of box corners (double, 2 element, 1-D array).

cornerlat

Latitude in decimal degrees (double) of box corners (double, 2 element, 1-D array).

mode

Cross Track inclusion mode (long). Allowable values are:

- 0 = Midpoint
- 1 = Endpoint
- 2 = Anypoint

Keywords

None

Examples

In this example, we define a region bounded by 3 degrees longitude, 5 degrees latitude and 7 degrees longitude, 12 degrees latitude. We will consider a cross track to be within the region if its midpoint is within the region:

```
cornerlon[0] = 3.d
cornerlat[0] = 5.d
cornerlon[1] = 7.d
cornerlat[1] = 12.d
regionID = EOS_SW_DEFBOXREGION(swathID, cornerlon, cornerlat, 0)
```

Version History

5.2	Introduced

EOS_SW_DEFCOMP

This function sets the HDF field compression for subsequent swath field definitions. The compression does not apply to one-dimensional fields. The compression schemes currently supported are: run length encoding (1), skipping Huffman (3), deflate (gzip) (4) and no compression (0, the default). Compressed fields are written using the standard EOS_SW_WRITEFIELD function, however, the entire field must be written in a single call. Any portion of a compressed field can then be accessed with the EOS_SW_READFIELD function. Compression takes precedence over merging so that multi-dimensional fields that are compressed are not merged. The user should refer to the HDF Reference Manual for a fuller explanation of the compression schemes and parameters.

Syntax

Result = EOS_SW_DEFCOMP(*swathID*, *compcode*, [, *compparm*])

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) otherwise.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

compcode

HDF compression code (long). Allowable values are:

- 0 = None
- 1 = Run Length Encoding (RLE)
- 3 = Skipping Huffman
- 4 = Deflate (gzip)

compparm

Deflate compression (*compcode* 4) requires a single integer compression parameter in the range of one to nine with higher values corresponding to greater compression.

Keywords

None

Examples

Suppose we wish to compress the Pressure using run length encoding, the Opacity field using deflate compression, the Spectra field with skipping Huffman compression, and use no compression for the Temperature field:

```
status = EOS_SW_DEFCOMP(swathID, 1)
status = EOS_SW_DEFDATAFIELD(swathID, "Pressure", $
   "Track,Xtrack", 5)
compparm[0] = 5
status = EOS_SW_DEFCOMP(swathID, 4, compparm)
status = EOS_SW_DEFDATAFIELD(swathID, "Opacity", $
   "Track,Xtrack", 5)
status = EOS_SW_DEFCOMP(swathID, 3)
status = EOS_SW_DEFDATAFIELD(swathID, "Spectra", $
   "Bands,Track,Xtrack", 5)
status = EOS_SW_DEFCOMP(swathID, 0)
status = EOS_SW_DEFDATAFIELD(swathID, $
   "Temperature", "Track,Xtrack", 5, /MERGE)
```

Note that the MERGE keyword will be ignored in the Temperature field definition.

Version History

5.2	Introduced

EOS_SW_DEFDATAFIELD

This function defines data fields to be stored in the swath. The dimensions are entered as a string consisting of data dimensions separated by commas. The API will attempt to merge into a single object those fields that share dimensions and in case of multidimensional fields, numbertype. If the merge keyword is not set, the API will not attempt to merge it with other fields. Because merging breaks the one-to-one correspondence between HDF-EOS fields and HDF SDS arrays, it should not be set if the user wishes to access the HDF-EOS field directly using HDF routines. To assure that the fields defined by EOS_SW_DEFDATAFIELD are properly established in the file, the swath should be detached (and then reattached) before writing to any fields.

Note

Array ordering of variables used or returned by this routine changed in IDL 5.5. Programs written for versions of this routine prior to IDL 5.5 may need to be modified to work correctly with the current version. See "Note on Array Ordering" on page 579 for details.

Syntax

Result = EOS_SW_DEFDATAFIELD(*swathID*, *fieldname*, *dimlist*, *numbertype* [, /MERGE])

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

fieldname

Name of field (string) to be defined.

dimlist

The list of data dimensions (string) defining the field.

numbertype

The HDF data type (long) of the data stored in the field.

Keywords

MERGE

If set, automatic merging will occur. By default, fields are not merged.

Examples

In this example, we define a three dimensional data field named Spectra with dimensions Bands, DataTrack, and DataXtrack:

```
status = EOS_SW_DEFDATAFIELD(swathID, "Spectra", $
   "Bands,DataTrack,DataXtrack", 5, /MERGE)
```

Version History

5.2	Introduced

EOS_SW_DEFDIM

This function defines dimensions that are used by the field definition functions (described subsequently) to establish the size of the field.

Syntax

Result = EOS_SW_DEFDIM(*swathID*, *fieldname, dim*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

swathID

Swath id (long).

fieldname

Name of dimension (string) to be defined.

dim

The size (long) of the dimension.

Keywords

None

Examples

In this example, we define a track geolocation dimension, GeoTrack, of size 2000, a cross track dimension, GeoXtrack, of size 1000 and two corresponding data dimensions with twice the resolution of the geolocation dimensions:

```
status = EOS_SW_DEFDIM(swathID, "GeoTrack", 2000)
status = EOS_SW_DEFDIM(swathID, "GeoXtrack", 1000)
status = EOS_SW_DEFDIM(swathID, "DataTrack", 4000)
status = EOS_SW_DEFDIM(swathID, "DataXtrack", 2000)
status = EOS_SW_DEFDIM(swathID, "Bands", 5)
```

To specify an unlimited dimension that can be used to define an appendable array, the dimension value should be set to zero:

```
status = EOS_SW_DEFDIM(swathID, "Unlim", 0)
```

Version History

5.2	Introduced

EOS_SW_DEFDIMMAP

This function defines monotonic mapping between the geolocation and data dimensions. Typically the geolocation and data dimensions are of different size (resolution). This function establishes the relation between the two where the offset gives the index of the data element (0-based) corresponding to the first geolocation element and the increment gives the number of data elements to skip for each geolocation element. If the geolocation dimension begins “before” the data dimension, then the offset is negative. Similarly, if the geolocation dimension has higher resolution than the data dimension, then the increment is negative. A typical reason for failure is an incorrect geolocation or data dimension name.

Syntax

Result = EOS_SW_DEFDIMMAP(*swathID*, *geodim*, *datadim*, *offset*, *increment*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

geodim

Geolocation dimension name (string).

datadim

Data dimension name (string).

offset

The offset (long) of the geolocation dimension with respect to the data dimension.

increment

The increment (long) of the geolocation dimension with respect to the data dimension.

Keywords

None

Examples

In this example, we establish the following:

- The first element of the GeoTrack dimension corresponds to the first element of the DataTrack dimension and the data dimension has twice the resolution of the geolocation dimension.
- The first element of the GeoXtrack dimension corresponds to the second element of the DataTrack dimension and the data dimension has twice the resolution of the geolocation dimension.

```
status=EOS_SW_DEFDIMMAP(swathID, "GeoTrack",  "DataTrack", 0, 2)
status=EOS_SW_DEFDIMMAP(swathID, "GeoXtrack", "DataXtrack", 1, 2)
```

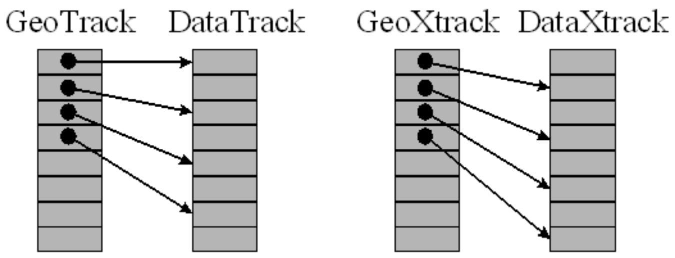


Figure 5-1:

Version History

5.2	Introduced

EOS_SW_DEFGEOFIELD

This function defines geolocation fields to be stored in the swath. The dimensions are entered as a string consisting of geolocation dimensions separated by commas. The API will attempt to merge into a single object those fields that share dimensions and in case of multidimensional fields, numbertype. If the merge keyword is not set, the API will not attempt to merge it with other fields. Fields using the unlimited dimension will not be merged. Because merging breaks the one-to-one correspondence between HDF-EOS fields and HDF SDS arrays, it should not be set if the user wishes to access the HDF field directly using HDF routines. To assure that the fields defined by EOS_SW_DEFGEOFIELD are properly established in the file, the swath should be detached (and then reattached) before writing to any fields.

Note

Array ordering of variables used or returned by this routine changed in IDL 5.5. Programs written for versions of this routine prior to IDL 5.5 may need to be modified to work correctly with the current version. See "Note on Array Ordering" on page 579 for details.

Syntax

Result = EOS_SW_DEFGEOFIELD(*swathID*, *fieldname*, *dimlist*, *numbertype* [, /MERGE])

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

fieldname

Name of field (string) to be defined.

dimlist

The list of geolocation dimensions (sting) defining the field.

numbertype

The HDF data type (long) of the data stored in the field.

Keywords

MERGE

If set, automatic merging will occur. By default, fields are not merged.

Examples

In this example, we define the geolocation fields, Longitude and Latitude with dimensions GeoTrack and GeoXtrack and containing 4 byte floating point numbers. We allow these fields to be merged into a single object:

```
status = EOS_SW_DEFGEOFIELD(swathID, "Longitude",$
   "GeoTrack,GeoXtrack", 5, /MERGE
status = EOS_SW_DEFGEOFIELD(swathID, "Latitude", $
   "GeoTrack,GeoXtrack", 5, /MERGE
```

Version History

5.2	Introduced

EOS_SW_DEFIDXMAP

If there does not exist a regular (linear) mapping between a geolocation and data dimension, then the mapping must be made explicit. Each element of the index array, whose dimension is given by the geolocation size, contains the element number (0-based) of the corresponding data dimension.

Syntax

Result = EOS_SW_DEFIDXMAP(*swathID*, *geodim*, *datadim*, *index*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

geodim

Geolocation dimension name (string).

datadim

Data dimension name (string).

index

The array (long) containing the indices of the data dimension to which each geolocation element corresponds.

Keywords

None

Examples

In this example, we consider the (simple) case of a geolocation dimension IdxGeo of size 5 and a data dimension IdxData of size 8. In this case, the 0th element of IdxGeo will correspond to the 0th element of IdxData, the 1st element of IdxGeo to the 2nd element of IdxData, etc.:

```
index = [0,2,3,6,7]
status = EOS_SW_DEFIDXMAP(swathID, "IdxGeo", "IdxData", index)
```

Version History

5.2	Introduced

EOS_SW_DEFTIMEPERIOD

This function defines a time period for a swath. It returns a swath period ID which is used by the EOS_SW_EXTRACTPERIOD function to read all the entries of a data field within the time period. A cross track is within a time period if its midpoint is within the time period box (0), or either of its endpoints is within the time period box (1), or any point of the cross track is within the time period box (2), depending on the inclusion mode designated by the user. All elements within an included cross track are considered to be within the time period even though a particular element of the cross track might be outside the time period. The swath structure must have the Time field defined.

Syntax

Result = EOS_SW_DEFTIMEPERIOD(*swathID*, *starttime* , *stoptime*, *mode*)

Return Value

Returns the swath period ID if successful and FAIL (–1) otherwise.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

starttime

Start time (double) of period.

stoptime

Stop time (double) of period.

mode

Cross Track inclusion mode (long). Allowable values are:

- 0 = Midpoint
- 1 = Endpoint
- 2 = Anypoint

Keywords

None

Examples

In this example, we define a time period with a start time of 35232487.2 and a stop time of 36609898.1.We will consider a cross track to be within the time period if either one of the time values at the endpoints of a cross track are within the time period:

```
starttime = 35232487.2d
stoptime = 36609898.1d
periodID = EOS_SW_DEFTIMEPERIOD(swathID, starttime, stoptime, 1)
```

Version History

5.2	Introduced

EOS_SW_DEFVRTREGION

Whereas the EOS_SW_DEFBOXREGION and EOS_SW_DEFTIMEPERIOD functions perform subsetting along the “Track” dimension, this function allows the user to subset along any dimension. The region is specified by a set of minimum and maximum values and can represent either a dimension index (case 1) or field value range (case 2). In the second case, the field must be one-dimensional and the values must be monotonic (strictly increasing or decreasing) in order that the resulting dimension index range be contiguous. (For the current version of this function, the second option is restricted to fields with one of the following HDF data types: 22, 24, 5, 6.)

This function may be called after EOS_SW_DEFBOXREGION or EOS_SW_DEFTIMEPERIOD to provide both geographic or time and “vertical” subsetting. In this case the user provides the id from the previous subset call. (This same id is then returned by the function.) This function may also be called “stand-alone” by setting the input id to (–1).

This function may be called up to eight times with the same region ID. It this way a region can be subsetted along a number of dimensions.

The EOS_SW_REGIONINFO and EOS_SW_EXTRACTREGION functions work as before, however, because there is no mapping performed between geolocation dimensions and data dimensions the field to be subsetted, (the field specified in the call to EOS_SW_REGIONINFO and EOS_SW_EXTRACTREGION) must contain the dimension used explicitly in the call to EOS_SW_DEFVRTREGION (case 1) or the dimension of the one-dimensional field (case 2).

Syntax

Result = EOS_SW_DEFVRTREGION(*swathID*, *regionID*, *vertObj*, *range*)

Return Value

Returns the swath region ID if successful and FAIL (–1) otherwise.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

regionID

Region or period id (long) from previous subset call, or –1 to create a new region within the entire dataset.

vertObj

Dimension or field (string) to subset by.

range

Minimum and maximum range (double) for subset.

Keywords

None

Examples

Suppose we have a field called Pressure of dimension Height whose values increase from 100 to1000, and we desire all the elements with values between 500 and 800:

```
range[0] = 500.d
range[1] = 800.d
regionID = EOS_SW_DEFVRTREGION(swathID, -1, "Pressure", range)
```

The function determines the elements in the Height dimension that correspond to the values of the Pressure field between 500 and 800.

If we wish to specify the subset as elements 2 through 5 (0 - based) of the Height dimension, the call would be:

```
range[0] = 2.d
range[1] = 5.d
regionID = EOS_SW_DEFVRTREGION(swathID, -1, "DIM:Height", range)
```

The “DIM:” prefix tells the routine that the range corresponds to elements of a dimension rather than values of a field. In this example, any field to be subsetted must contain the Height dimension.

If a previous subset region or period was defined with an id of subsetID that we wish to refine further with the vertical subsetting defined above, we make the call:

```
regionID = EOS_SW_DEFVRTREGION(swathID, subsetID, $
   "Pressure", range)
```

The return value, regionID, is set equal to subsetID. That is, the subset region is modified rather than a new one created. We can further refine the subset region with another call to the function:

```
freq[0] = 1540.3d
freq[1] = 1652.8d
regionID = EOS_SW_DEFVRTREGION(swathID, regionID, $
   "FreqRange", freq)
```

Version History

5.2	Introduced

EOS_SW_DETACH

This function detaches from the swath interface. It should be run before exiting from the swath file for every swath opened by EOS_SW_CREATE or EOS_SW_ATTACH.

Syntax

Result = EOS_SW_DETACH(*swathID*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

Keywords

None

Examples

```
status = EOS_SW_DETACH(swathID)
```

Version History

5.2	Introduced

See Also

EOS_SW_ATTACH

EOS_SW_DIMINFO

This function retrieves the size of the specified dimension.

Syntax

Result = EOS_SW_DIMINFO(*swathID*, *dimname*)

Return Value

Size of dimension or FAIL (–1) if the swath ID or dimension name are invalid.

Arguments

swathID

Swath id (long).

dimname

Dimension name (string).

Keywords

None

Examples

In this example, we retrieve information about the dimension, “GeoTrack”:

```
dimsize = EOS_SW_DIMINFO(swathID, "GeoTrack")
```

Version History

5.2	Introduced

EOS_SW_DUPREGION

This function copies the information stored in a current region or period to a new region or period and generates a new id. It is useful when the user wishes to further subset a region (period) in multiple ways.

Syntax

Result = EOS_SW_DUPREGION(*regionID*)

Return Value

Returns new region or period ID or FAIL (–1) on error.

Arguments

regionID

Region or period id (long) returned by EOS_SW_DEFBOXREGION, EOS_SW_DEFTIMEPERIOD, or EOS_SW_DEFVRTREGION.

Keywords

None

Examples

In this example, we first subset a swath with EOS_SW_DEFBOXREGION, duplicate the region creating a new region ID, regionID2, and then perform two different vertical subsets of these (identical) geographic subset regions:

```
regionID = EOS_SW_DEFBOXREGION(swathID, cornerlon, $
   cornerlat, 0)
regionID2 = EOS_SW_DUPREGION(regionID)
regionID = EOS_SW_DEFVRTREGION(swathID, regionID, $
   "Pressure", rangePres)
regionID2 = EOS_SW_DEFVRTREGION(swathID, regionID2, $
   "Temperature", rangeTemp)
```

Version History

5.2	Introduced

EOS_SW_EXTRACTPERIOD

This function reads data into the data buffer from the subsetted time period. Only complete crosstracks are extracted. If the external_mode flag is set to (1) then the geolocation fields and the data field can be in different swaths. If set to (0), then these fields must be in the same swath structure.

Note

Array ordering of variables used or returned by this routine changed in IDL 5.5. Programs written for versions of this routine prior to IDL 5.5 may need to be modified to work correctly with the current version. See "Note on Array Ordering" on page 579 for details.

Syntax

Result = EOS_SW_EXTRACTPERIOD(*swathID*, *periodID*, *fieldname*, *external_mode*, *buffer*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

swathID

Swath id (long).

periodID

Period id (long) returned by EOS_SW_DEFTIMEPERIOD.

fieldname

Field to subset (string).

external_mode

External geolocation mode (long).

buffer

A named variable that will contain the period data.

Keywords

None

Examples

In this example, we read data within the subsetted time period defined in EOS_SW_DEFTIMEPERIOD from the Spectra field. Both the geolocation fields and the Spectra data field are in the same swath.

```
status = EOS_SW_EXTRACTPERIOD(EOS_SW_id, periodID, 0,"Spectra", $
  datbuf)
```

Version History

5.2	Introduced

EOS_SW_EXTRACTREGION

This function reads data into the data buffer from the subsetted region. Only complete crosstracks are extracted. If the external_mode flag is set to (1) then the geolocation fields and the data field can be in different swaths. If set to (0), then these fields must be in the same swath structure.

Note

Array ordering of variables used or returned by this routine changed in IDL 5.5. Programs written for versions of this routine prior to IDL 5.5 may need to be modified to work correctly with the current version. See "Note on Array Ordering" on page 579 for details.

Syntax

Result = EOS_SW_EXTRACTREGION(*swathID*, *regionID*, *fieldname*, *external_mode*, *buffer*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

regionID

Region id (long) returned by EOS_SW_DEFBOXREGION.

fieldname

Field to subset (string).

external_mode

External geolocation mode (long).

buffer

A named variable that will contain the data buffer.

Keywords

None

Examples

In this example, we read data within the subsetted region defined in EOS_SW_DEFBOXREGION from the Spectra field. Both the geolocation fields and the Spectra data field are in the same swath.

```
status = EOS_SW_EXTRACTREGION(EOS_SW_id, regionID, 0, "Spectra",$
   datbuf)
```

Version History

5.2	Introduced

EOS_SW_FIELDINFO

This function retrieves information on a specific data field.

Syntax

Result = EOS_SW_FIELDINFO(*swathID*, *fieldname*, *rank*, *dims*, *numbertype*, *dimlist*)

Return Value

Returns SUCCEED(0) if successful and FAIL(–1) if the specified field does not exist.

Arguments

swathID

Swath id (long).

fieldname

Fieldname (string).

rank

A named variable that will contain the rank of field (long).

dims

A named variable that will contain the array of length “rank” (long) containing the dimension sizes of the field. If one of the dimensions in the field is appendable, then the current value for that dimension will be returned in the dims array.

numbertype

A named variable that will contain HDF data type of the field.

dimlist

A named variable that will contain the list of dimensions (string) in field.

Keywords

None

Examples

In this example, we retrieve information about the Spectra data fields:

```
status = EOS_SW_FIELDINFO(swathID, "Spectra", rank, dims, $
   numbertype, dimlist)
```

Version History

5.2	Introduced

EOS_SW_GETFILLVALUE

This function retrieves the fill value for the specified field.

Syntax

Result = EOS_SW_GETFILLVALUE(*swathID*, *fieldname*, *fillvalue*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

fieldname

Fieldname (string).

fillvalue

A named variable that will contain the fill value.

Keywords

None

Examples

In this example, we get the fill value for the “Temperature” field:

```
status = EOS_SW_GETFILLVALUE(swathID, "Temperature", tempfill)
```

Version History

5.2	Introduced

EOS_SW_IDXMAPINFO

This function retrieves the size of the indexed array and the array of indexed elements of the specified geolocation mapping.

Syntax

Result = EOS_SW_IDXMAPINFO(*swathID*, *geodim*, *datadim*, *index*)

Return Value

Returns size of indexed array if successful and FAIL (–1) if the specified mapping does not exist.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

geodim

Geolocation dimension name (string).

datadim

Data dimension name (string).

index

A named variable that will contain an array (long) of indices of the data dimension to which each geolocation element corresponds.

Keywords

None

Examples

In this example, we retrieve information about the indexed mapping between the “IdxGeo” and “IdxData” dimensions:

```
idxsz = EOS_SW_IDXMAPINFO(swathID, "IdxGeo", "IdxData", index)
```

Version History

5.2	Introduced

EOS_SW_INQATTRS

This function retrieves information about attributes defined in swath. The attribute list is returned as a string with each attribute name separated by commas.

Note
See STRSPLIT to separate the attribute list.

Syntax

Result = EOS_SW_INQATTRS(*swathID*, *attrlist* [, LENGTH=*variable*])

Return Value

Number of attributes found or (–1) if failure.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

attrlist

A named variable that will contain the attribute list (string) with entries separated by commas.

Keywords

LENGTH

Set this keyword to a named variable that will contain the length of the attribute list as a long integer.

Examples

```
nattr = EOS_SW_INQATTRS(swathID, attrlist)
```

Version History

5.2	Introduced

EOS_SW_INQDATAFIELDS

This function retrieves information about all of the data fields defined in swath. The field list is returned as a string with each data field separated by commas. The rank and numbertype arrays will have an entry for each field.

Note

See STRSPLIT to separate the field list.

Syntax

Result = EOS_SW_INQDATAFIELDS(*swathID*, *fieldlist*, *rank*, *numbertype*)

Return Value

Returns number of data fields found. If –1, could signify improper swath id.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

fieldlist

A named variable that will contain the listing of data fields (string) with entries separated by commas.

rank

A named variable that will contain an array (long) of rank of each data field.

numbertype

A named variable that will contain an array (long) of numbertype of each data field.

Keywords

None

Examples

```
nflds = EOS_SW_INQDATAFIELDS(swathID, fieldlist, rank, numbertype)
```

Version History

5.2	Introduced

EOS_SW_INQDIMS

This function retrieves information about all of the dimensions defined in swath. The dimension list is returned as a string with each dimension name separated by commas.

Note

See STRSPLIT to separate the dimension list.

Syntax

Result = EOS_SW_INQDIMS(*swathID*, *dimname*, *dim*)

Return Value

Returns number of dimension entries found. If –1, could signify an improper swath id.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

dimname

A named variable that will contain the dimension list (string) with entries separated by commas.

dims

A named variable that will contain an array (long) of size of each dimension.

Keywords

None

Examples

```
ndims = EOS_SW_INQDIMS(swathID, dimname, dims)
```

Version History

5.2	Introduced

EOS_SW_INQGEOFIELDS

This function retrieves information about all of the geolocation fields defined in swath. The field list is returned as a string with each geolocation field separated by commas. The rank and numbertype arrays will have an entry for each field.

Note

See STRSPLIT to separate the field list.

Syntax

Result = EOS_SW_INQGEOFIELDS(*swathID*, *fieldlist*, *rank*, *numbertype*)

Return Value

Returns number of geolocation fields found. If –1, could signify an improper swath id.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

fieldlist

A named variable that will contain the listing of geolocation fields (string) with entries separated by commas.

rank

A named variable that will contain an array (long) of the rank of each geolocation field.

numbertype

A named variable that will contain an array (long) of the numbertype of each geolocation field.

Keywords

None

Examples

```
nflds = EOS_SW_INQGEOFIELDS(swathID, fieldlist, rank, numbertype)
```

Version History

5.2	Introduced

EOS_SW_INQIDXMAPS

This function retrieves information about all of the indexed geolocation/data mappings defined in swath. The dimension mapping list is returned as a string with each mapping separated by commas. The two dimensions in each mapping are separated by a slash (/).

Note

See STRSPLIT to separate the mapping list.

Syntax

Result = EOS_SW_INQIDXMAPS(*swathID*, *idxmap*, *idxsizes*)

Return Value

Number of indexed mapping relations found. If –1, could signify an improper swath id.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

idxmap

A named variable that will contain the indexed Dimension mapping list (string) with entries separated by commas.

idxsizes

A named variable that will contain an array (long) of the sizes of the corresponding index arrays.

Keywords

None

Examples

```
nidxmaps = EOS_SW_INQIDXMAPS(swathID, idxmap, idxsizes)
```

Version History

5.2	Introduced

EOS_SW_INQMAPS

This function retrieves information about all of the (non-indexed) geolocation relations defined in swath. The dimension mapping list is returned as a string with each mapping separated by commas. The two dimensions in each mapping are separated by a slash (/).

Note

See STRSPLIT to separate the mapping list.

Syntax

Result = EOS_SW_INQMAPS(*swathID*, *dimmap*, *offset*, *increment*)

Return Value

Number of geolocation relation entries found. If –1, could signify an improper swath id.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

dimmap

A named variable that will contain the dimension mapping list (string) with entries separated by commas.

offset

A named variable that will contain an array (long) of the offset of each geolocation relation.

increment

A named variable that contain an array (long) of the increment of each geolocation relation.

Keywords

None

Examples

```
nmaps = EOS_SW_INQMAPS(swathID, dimmap, offset, increment)
```

Version History

5.2	Introduced

EOS_SW_INQSWATH

This function retrieves number and names of swaths defined in the HDF-EOS file. The swath list is returned as a string with each swath name separated by commas.

Note

See STRSPLIT to separate the swath list.

Syntax

Result = EOS_SW_INQSWATH(*filename*, *swathlist* [, LENGTH =*value*])

Return Value

Number of swaths found or (–1) if failure.

Arguments

filename

HDF-EOS filename (string).

swathlist

Swath list (string) with entries separated by commas.

Keywords

LENGTH

String length (long) of swath list.

Examples

In this example, we retrieve information about the swaths defined in an HDF-EOS file, HDFEOS.hdf:

```
nswath = EOS_SW_INQSWATH("HDFEOS.hdf", swathlist)
```

Version History

5.2	Introduced

EOS_SW_MAPINFO

This function retrieves the offset and increment of the specified geolocation mapping.

Syntax

Result = EOS_SW_MAPINFO(*swathID*, *geodim*, *datadim*, *offset*, *increment*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) if the specified mapping does not exist.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

geodim

Geolocation dimension name (string).

datadim

Data dimension name (string).

offset

A named variable that will contain the mapping offset (long).

increment

A named variable that will contain the mapping increment (long).

Keywords

None

Examples

In this example, we retrieve information about the mapping between the GeoTrack and DataTrack dimensions:

```
status = EOS_SW_MAPINFO(swathID, "GeoTrack", "DataTrack", $
   offset, increment)
```

Version History

5.2	Introduced

EOS_SW_NENTRIES

This function returns number of entries and descriptive string buffer size for a specified entity. This function can be called before an inquiry routine in order to determine the sizes of the output arrays and descriptive strings.

Syntax

Result = EOS_SW_NENTRIES(*swathID*, *entrycode* [, LENGTH=*variable*])

Return Value

Number of entries or FAIL (–1) in the case of an improper swath id or entry code.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

entrycode

Entrycode (long). Allowable values are:

- 0 = Dimensions
- 1 = Dimension Mappings
- 2 = Indexed Dimension Mappings
- 3 = Geolocation Fields
- 4 = Data Fields

Keywords

LENGTH

Set this keyword to a named variable that will contain the length of the string that would be returned by the corresponding inquiry routine, as a long integer.

Examples

In this example, we determine the number of dimension mapping entries.

```
nmaps = EOS_SW_NENTRIES(swathID, 2)
```

Version History

5.2	Introduced

EOS_SW_OPEN

This function creates a new file or opens an existing file.

Syntax

Result = EOS_SW_OPEN(*filename* [, /CREATE] [, /RDWR | , /READ])

Return Value

Returns the swath file id handle (fid) if successful and FAIL (–1) otherwise.

Arguments

filename

Complete path and filename for the file to be opened (string).

Keywords

CREATE

If file exists, delete it, then open a new file for read/write.

RDWR

Open for read/write, If file does not exist, create it.

READ

Open for read only. If file does not exist, error. This is the default.

Examples

In this example, we create a new swath file named, SwathFile.hdf. It returns the file handle, fid:

```
fid = EOS_SW_OPEN("SwathFile.hdf", /CREATE)
```

Version History

5.2	Introduced

EOS_SW_PERIODINFO

This function returns information about a subsetted time period for a particular field. Because of differences in number type and geolocation mapping, a given time period will give different values for the dimensions and size for various fields.

Note

Array ordering of variables used or returned by this routine changed in IDL 5.5. Programs written for versions of this routine prior to IDL 5.5 may need to be modified to work correctly with the current version. See "Note on Array Ordering" on page 579 for details.

Syntax

Result = EOS_SW_PERIODINFO(*swathID*, *periodID*, *fieldname*, *ntype*, *rank*, *dims*, *size*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

periodID

Period id (long) returned by EOS_SW_DEFTIMEPERIOD.

fieldname

Field to subset (string).

ntype

A named variable that will contain the number type of field (long).

rank

A named variable that will contain the rank of field (long).

dims

A named variable that will contain the dimensions of subset period (long).

size

A named variable that will contain the size in bytes of subset period (long).

Keywords

None

Examples

In this example, we retrieve information about the time period defined in EOS_SW_DEFTIMEPERIOD for the Spectra field:

```
; Get size in bytes of time period for "Spectra" field
status = EOS_SW_PERIODINFO(EOS_SW_id, periodID, $
   "Spectra", ntype, rank, dims, size)
```

Version History

5.2	Introduced

EOS_SW_QUERY

The EOS_SW_QUERY function returns information about a specified swath.

Syntax

Result=EOS_SW_QUERY(*Filename*, *SwathName*, [*Info*])

Return Value

This function returns an integer value of 1 if the file is an HDF file with EOS SWATH extensions, and 0 otherwise.

Arguments

Filename

A string containing the name of the file to be queried.

SwathName

A string containing the name of the swath to be queried.

Info

Returns an anonymous structure containing information about the specified swath. The returned structure contains the following fields:

Field	IDL data type	Description
ATTRIBUTES	String array	Array of attribute names
DIMENSION_NAMES	String array	Names of dimensions
DIMENSION_SIZES	Long array	Sizes of dimensions
FIELD_NAMES	String array	Names of fields
FIELD_RANKS	Long array	Ranks (dimensions) of fields
FIELD_TYPES	Long array	IDL types of fields

Table 5-4: Fields of the Info Structure

Field	IDL data type	Description
GEO_FIELD_NAMES	String array	Names of geolocation fields
GEO_FIELD_RANKS	Long array	Ranks (dimensions) of geolocation fields
GEO_FIELD_TYPES	Long array	IDL types of geolocation fields
IDX_MAP_NAMES	String array	Names of index maps
IDX_MAP_SIZES	Long array	Sizes of index map arrays
NUM_ATTRIBUTES	Long	Number of attributes
NUM_DIMS	Long	Number of dimensions
NUM_FIELDS	Long	Number of fields
NUM_GEO_FIELDS	Long	Number of geolocation fields
NUM_IDX_MAPS	Long	Number of indexed dimension mapping entries
NUM_MAPS	Long	Number of mapping entries
MAP_INCREMENTS	Long array	Increment of each geolocation relation
MAP_NAMES	String array	Names of maps
MAP_OFFSETS	Long array	Offset of each geolocation relation

Table 5-4: Fields of the Info Structure (Continued)

Keywords

None

Version History

5.3	Introduced

EOS_SW_READATTR

This function reads attributes from a swath field.

Syntax

Result = EOS_SW_READATTR(*swathID*, *attrname*, *datbuf*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

attrname

Attribute name (string).

datbuf

A named variable that will contain the attribute values.

Keywords

None

Examples

In this example, we read a single precision (32-bit) floating-point attribute with the name “ScalarFloat”:

```
status = EOS_SW_READATTR(swathID, "ScalarFloat", f32)
```

Version History

5.2	Introduced

EOS_SW_READFIELD

This function reads data from a swath field. The values within start, stride, and edge arrays refer to the swath field (input) dimensions. The default values for start and stride are 0 and 1 respectively if these keywords are not set. The default value for edge is (dim – start) / stride where dim refers to the IDL variable dimension.

Note

Array ordering of variables used or returned by this routine changed in IDL 5.5. Programs written for versions of this routine prior to IDL 5.5 may need to be modified to work correctly with the current version. See "Note on Array Ordering" on page 579 for details.

Syntax

Result = EOS_SW_READFIELD(*swathID*, *fieldname*, *buffer* [, EDGE=*array*]
[, START=*array*] [, STRIDE=*array*])

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

fieldname

Name of field to read (string).

buffer

A named variable that will contain the data read from the field.

Keywords

EDGE

Array (long) specifying the number of values to read along each dimension.

START

Array (long) specifying the starting location within each dimension.

STRIDE

Set this keyword to an array of integers specifying the number of values to step along each dimension. The default is [1, 1, ...] indicating that every value should be included. Specifying a stride of 0 is equivalent to 1.

Examples

In this example, we read data from the 10th track (0-based) of the Longitude field:

```
start=[10,1]
edge=[1,1000]
status = EOS_SW_READFIELD(swathID, "Longitude", track, $
   START = start, EDGE = edge)
```

Version History

5.2	Introduced

EOS_SW_REGIONINFO

This function returns information about a subsetted region for a particular field. Because of differences in number type and geolocation mapping, a given region will give different values for the dimensions and size for various fields.

Note

Array ordering of variables used or returned by this routine changed in IDL 5.5. Programs written for versions of this routine prior to IDL 5.5 may need to be modified to work correctly with the current version. See "Note on Array Ordering" on page 579 for details.

Syntax

Result = EOS_SW_REGIONINFO(*swathID*, *regionID*, *fieldname*, *ntype*, *rank*, *dims*, *size*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

regionID

Region id (long) returned by EOS_SW_DEFBOXREGION.

fieldname

Field to subset (string).

ntype

A named variable that will contain the number type of field (long).

rank

A named variable that will contain the rank of field (long).

dims

A named variable that will contain the dimensions of subset region (long).

size

A named variable that will contain the size in bytes of subset region (long).

Keywords

None

Examples

In this example, we retrieve information about the region defined in EOS_SW_DEFBOXREGION for the Spectra field:

```
status = EOS_SW_REGIONINFO(EOS_SW_id, regionID, "Spectra", $
   ntype, rank, dims, size)
```

Version History

5.2	Introduced

EOS_SW_SETFILLVALUE

This function sets the fill value for the specified field. The fill value is placed in all elements of the field that have not been explicitly defined.

Syntax

Result = EOS_SW_SETFILLVALUE(*swathID*, *fieldname*, *fillvalue*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW__ATTACH.

fieldname

Fieldname (string).

fillvalue

The fill value to be used.

Keywords

None

Examples

In this example, we set a fill value for the “Temperature” field:

```
tempfill = -999.0
status = EOS_SW_SETFILLVALUE(swathID, "Temperature", tempfill)
```

Version History

5.2	Introduced

EOS_SW_WRITEATTR

This function writes/updates attributes in a swath. If the attribute does not exist, it is created. If it does exist, then the value is updated.

Syntax

Result = EOS_SW_WRITEATTR(*swathID*, *attrname*, *datbuf* [, COUNT=*value*] [, HDF_TYPE=*value*])

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

attrname

Attribute name (string).

datbuf

Attribute values (long). If HDF_TYPE is specified, the attribute values are first converted to the type specified by HDF_TYPE before being stored.

Keywords

COUNT

Number of values to store in attribute (long).

HDF_TYPE

HDF data type of the attribute. See “IDL and HDF Data Types” on page 317 for valid values.

Examples

In this example, we write a single precision (32 bit) floating point number with the name “ScalarFloat” and the value 3.14:

```
f32 = 3.14
status = EOS_SW_WRITEATTR(swathid, "ScalarFloat", f32)
```

We can update this value by simply calling the function again with the new value:

```
f32 = 3.14159
status = EOS_SW_WRITEATTR(swathid, "ScalarFloat", f32)
```

Version History

5.2	Introduced

EOS_SW_WRITEDATAMETA

This function writes field metadata for an existing data field. This is useful when the data field was defined without using the swath API. Note that any entries in the dimension list must be defined through the EOS_SW_DEFDIM function before this function is called.

Note

Array ordering of variables used or returned by this routine changed in IDL 5.5. Programs written for versions of this routine prior to IDL 5.5 may need to be modified to work correctly with the current version. See "Note on Array Ordering" on page 579 for details.

Syntax

Result = EOS_SW_WRITEDATAMETA(*swathID*, *fieldname*, *dimlist*, *numbertype*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

fieldname

Name of field (string).

dimlist

The list of data dimensions defining the field (string).

numbertype

The number type of the data stored in the field (long).

Keywords

None

Examples

In this example, we write the metadata for the “Band_1” data field used in the swath:

```
status = EOS_SW_WRITEDATAMETA(swathID, "Band_1", $
   "GeoTrack,GeoXtrack",5)
```

Version History

5.2	Introduced

EOS_SW_WRITEFIELD

This function writes data to a swath field. The values within start, stride, and edge arrays refer to the swath field (output) dimensions. The default values for start and stride are 0 and 1 respectively and are used if keywords are not set. The default value for edge is (dim – start) / stride where dim refers to the size of the dimension. Note that the data buffer for a compressed field must be the size of the entire field as incremental writes are not supported by the underlying HDF routines.

Note

Array ordering of variables used or returned by this routine changed in IDL 5.5. Programs written for versions of this routine prior to IDL 5.5 may need to be modified to work correctly with the current version. See "Note on Array Ordering" on page 579 for details.

Syntax

Result = EOS_SW_WRITEFIELD(*swathID*, *fieldname*, *data* [, EDGE=*array*] [, START=*array*] [, STRIDE=*array*])

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

fieldname

Name of field to write (string).

data

Values to be written to the field.

Keywords

EDGE

Array (long) specifying the number of values to write along each dimension.

START

Array (long) specifying the starting location within each dimension (0-based).

STRIDE

Set this keyword to an array of integers specifying the number of values to step along each dimension. The default is [1, 1, ...] indicating that every value should be included. Specifying a stride of 0 is equivalent to 1.

Examples

In this example, we write data to the Longitude field:

```
; Define elements of longitude array:
longitude = indgen(2000, 1000)
status = EOS_SW_WRITEFIELD(swathID,"Longitude", longitude)
; We now update Track 10 (0 - based) in this field:
newtrack = intarr (1,1000)
start=[10,0]
edge =[1,1000]
; Define elements of newtrack array:
status = EOS_SW_WRITEFIELD(swathID, "Longitude",newtrack, $
   START = start, EDGE = edge)
```

Version History

5.2	Introduced

EOS_SW_WRITEGEOMETA

This function writes field metadata for an existing geolocation field. This is useful when the data field was defined without using the swath API. Note that any entries in the dimension list must be defined through the EOS_SW_DEFDIM function before this function is called.

Note

Array ordering of variables used or returned by this routine changed in IDL 5.5. Programs written for versions of this routine prior to IDL 5.5 may need to be modified to work correctly with the current version. See "Note on Array Ordering" on page 579 for details.

Syntax

Result = EOS_SW_WRITEGEOMETA(*swathID*, *fieldname*, *dimlist*, *numbertype*)

Return Value

Returns SUCCEED (0) if successful and FAIL (–1) otherwise.

Arguments

swathID

Swath id (long) returned by EOS_SW_CREATE or EOS_SW_ATTACH.

fieldname

Name of field (string).

dimlist

The list of geolocation dimensions (string) defining the field.

numbertype

The number type of the data (long) stored in the field.

Keywords

None

Examples

In this example, we write the metadata for the “Latitude” geolocation field used in the swath:

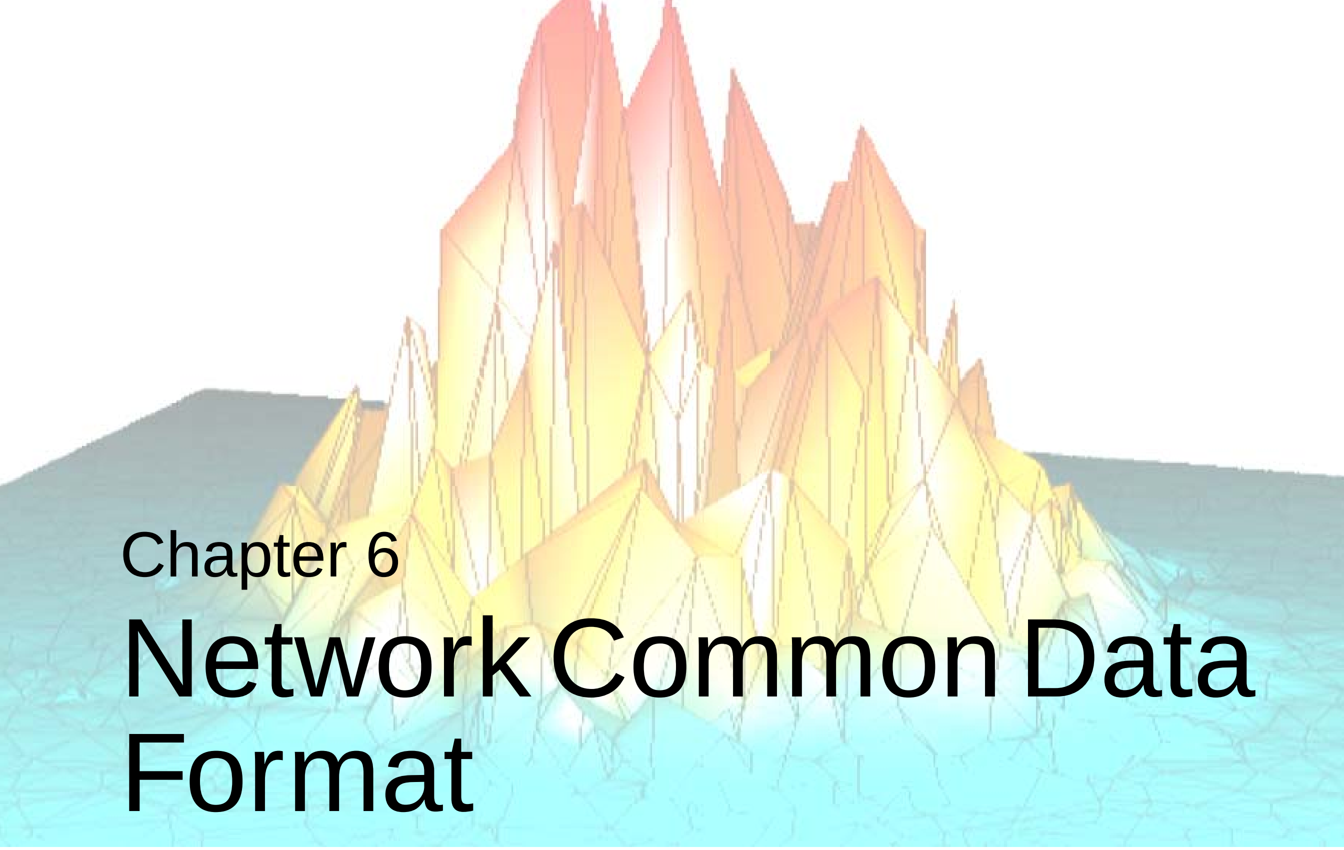
```
status = EOS_SW_WRITEGEOMETA(swathID, $
   "Latitude", "GeoTrack,GeoXtrack", 5)
```

Version History

5.2	Introduced

Keywords

None



Chapter 6
Network Common Data Format

The following topics are covered in this appendix:

Overview of NetCDF

The network Common Data Format (netCDF) is a self-describing scientific data access interface and library developed at the Unidata Program Center in Boulder, Colorado. The netCDF interface and library use XDR (eXternal Data Representation) to make the data format machine-independent. This version of IDL supports netCDF 3.5. IDL's NetCDF routines all begin with the prefix "NCDF_".

More information about netCDF can be found on Unidata's netCDF World Wide Web home page which can be found at:

`http://www.unidata.ucar.edu/packages/netcdf/`

Further information and the original netCDF documentation can be obtained from Unidata at the following addresses:

UCAR Unidata Program Center
P.O. Box 3000
Boulder, Colorado, USA 80307
(303) 497-8644
e-mail: support@unidata.ucar.edu

NetCDF Data Modes

There are two modes associated with accessing a netCDF file: *define* mode and *data* mode. In define mode, dimensions, variables, and new attributes can be created but variable data cannot be read or written. In data mode, data can be read or written and attributes can be changed, but new dimensions, variables, and attributes cannot be created.

IDL's NCDF_CONTROL routine can be used control the mode of a netCDF file. The only time it is not necessary to set the mode with NCDF_CONTROL is when using the NCDF_CREATE procedure to create a new file. NCDF_CREATE places the new netCDF file into define mode automatically.

Attributes, Dimensions, and Variables

The three basic components of a netCDF file are described below.

Attributes

Attributes can contain auxiliary information about an entire netCDF file (*global* attributes) or about a single netCDF variable. Every attribute has a name, data type, and length associated with it. It is common to repeat attribute names for each variable. For example, every variable in a netCDF file might have an attribute named “Units”. Note however, that variables cannot have multiple attributes with the same names.

Dimensions

Dimensions are named integers that are used to specify the size (or *dimensionality*) of one or more variables. Each dimension must have a unique name, but a variable and dimension can share a name. Each netCDF file is allowed to have one boundless (or *unlimited*) dimension. Most often the unlimited dimension is used as a temporal variable, allowing data to be appended to an existing netCDF file. An example of this use is shown later.

Variables

Variables are multidimensional arrays of values of the same data type. Each variable has a size, type, and name associated with it. Variables can also have *attributes* that describe them.

Creating NetCDF Files

The following IDL commands should be used to create a new netCDF file:

- NCDF_CREATE: Call this procedure to begin creating a new file. The new file is put into *define* mode.
- NCDF_DIMDEF: Create dimensions for the file.
- NCDF_VARDEF: Define the variables to be used in the file.
- NCDF_ATTPUT: Optionally, use attributes to describe the data.
- NCDF_CONTROL, /ENDEF: Call NCDF_CONTROL and set the ENDEF keyword to leave *define* mode and enter *data* mode.
- NCDF_VARPUT: Write the appropriate data to the netCDF file.
- NCDF_CLOSE: Close the file.

Reading NetCDF Files

The following commands should be used to read data from a netCDF file:

- NCDF_OPEN: Open an existing netCDF file.
- NCDF_INQUIRE: Call this function to find the format of the netCDF file.
- NCDF_DIMINQ: Retrieve the names and sizes of dimensions in the file.
- NCDF_VARINQ: Retrieve the names, types, and sizes of variables in the file.
- NCDF_ATTNAME: Optionally, retrieve attribute names.
- NCDF_ATTINQ: Optionally, retrieve the types and lengths of attributes.
- NCDF_ATTGET: Optionally, retrieve the attributes.
- NCDF_VARGET: Read the data from the variables.
- NCDF_CLOSE: Close the file.

If the structure of the netCDF file is already known, the inquiry routines do not need to be called—only NCDF_OPEN, NCDF_ATTGET, NCDF_VARGET, and NCDF_CLOSE would be needed.

NetCDF Examples

Example Code

Two example files that demonstrate the use of the netCDF routines can be found in the `examples/doc/sdf` subdirectory of the IDL distribution. The file `ncdf_cat.pro` prints a summary of basic information about a netCDF file. The file `ncdf_rdwr.pro` creates a new netCDF file and then reads the information back from that file.

A Complete Example with Unlimited Dimensions

The following example shows how to create a netCDF file, populate it with data, read data from the file, and make a simple plot from the data.The resulting graphic is shown below.

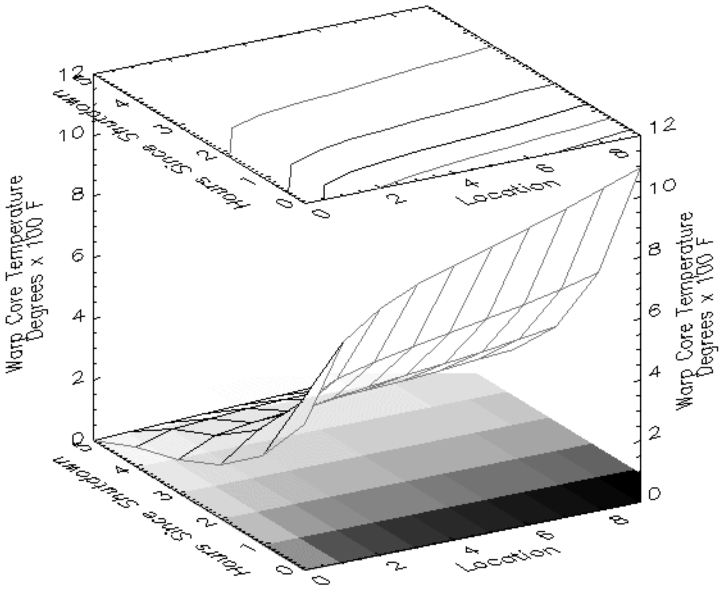


Figure 6-1: SHOW3 result of unlimited dimensions example

```
; Create a new NetCDF file with the filename inquire.nc:
id = NCDF_CREATE('inquire.nc', /CLOBBER)
; Fill the file with default values:
NCDF_CONTROL, id, /FILL
; We’ll create some time-dependent data, so here is an
; array of hours from 0 to 5:
hours = INDGEN(5)
; Create a 5 by 10 array to hold floating-point data:
data = FLTARR(5,10)
; Generate some values.
FOR i=0,9 DO $
   data(*,i) = (i+0.5) * EXP(-hours/2.) / SIN((i+1)/30.*!PI)
xid = NCDF_DIMDEF(id, 'x', 10)    ; Make dimensions.
zid = NCDF_DIMDEF(id, 'z', /UNLIMITED)
; Define variables:
hid = NCDF_VARDEF(id, 'Hour', [zid], /SHORT)
vid = NCDF_VARDEF(id, 'Temperature', [xid,zid], /FLOAT)
NCDF_ATTPUT, id, vid, 'units', 'Degrees x 100 F'
NCDF_ATTPUT, id, vid, 'long_name', 'Warp Core Temperature'
NCDF_ATTPUT, id, hid, 'long_name', 'Hours Since Shutdown'
NCDF_ATTPUT, id, /GLOBAL, 'Title', 'Really important data'
; Put file in data mode:
NCDF_CONTROL, id, /ENDEF
; Input data:
NCDF_VARPUT, id, hid, hours
FOR i=0,4 DO NCDF_VARPUT, id, vid, $
; Oops! We forgot the 6th hour! This is not a problem, however,
; as you can dynamically expand a netCDF file if the unlimited
; dimension is used.
   REFORM(data(i,*)), OFFSET=[0,i]
; Add the hour and data:
NCDF_VARPUT, id, hid, 6, OFFSET=[5]
; Add the temperature:
NCDF_VARPUT, id, vid, FINDGEN(10)*EXP(-6./2), OFFSET=[0,5]
; Read the data back out:
NCDF_VARGET, id, vid, output_data
NCDF_ATTGET, id, vid, 'long_name', ztitle
NCDF_ATTGET, id, hid, 'long_name', ytitle
NCDF_ATTGET, id, vid, 'units', subtitle
!P.CHARSIZE = 2.5
!X.TITLE = 'Location'
!Y.TITLE = STRING(ytitle) ; Convert from bytes to strings.
!Z.TITLE = STRING(ztitle) + '!C' + STRING(subtitle)
NCDF_CLOSE, id ; Close the NetCDF file.
SHOW3, output_data ; Display the data.
```

Type Conversion

Values are converted to the appropriate type before being written to a netCDF file. For example, in the commands below, IDL converts the string “12” to a floating-point 12.0 before writing it:

```
varid=NCDF_VARDEF(fileid, 'VarName', [d0,d1,d2+d3], /FLOAT)
NCDF_VARPUT, fileid, 'VarName', '12'
```

Specifying Attributes and Variables

Variables and attributes can be referred to either by name or by their ID numbers in most netCDF routines. For example, given the NCDF_VARDEF command shown below, the two NCDF_VARPUT commands shown after it are equivalent:

```
varid = NCDF_VARDEF(fileid, 'VarName', [d0,d1,d2+d3], /FLOAT)
; Reference by variable name:
NCDF_VARPUT, fileid, 'VarName', '12'
; Reference by variable ID:
NCDF_VARPUT, fileid, varid,'12'
```

String Data in NetCDF Files

Strings are stored as arrays of ASCII bytes in netCDF files. To read string data from netCDF files, use the STRING function to convert bytes back into characters. When writing an IDL string array to a variable, an extra dimension (the maximum string length) must be added to the variable definition. Both of these situations are illustrated by the following example:

```
; Make a test string:
string_in = REPLICATE('Test String',10,10)
; Make one element longer than the others:
string_in(0,0) = 'Long Test String'
HELP, string_in
; Create a new netCDF file:
ncdfid = NCDF_CREATE('string.nc', /CLOBBER)
; Define first dimension:
xid = NCDF_DIMDEF(ncdfid, 'height', 10)
; Define second dimension:
yid = NCDF_DIMDEF(ncdfid, 'width', 10)
; Find the length of the longest string and use that as the
; third dimension:
zid = NCDF_DIMDEF(ncdfid, 'length', MAX(STRLEN(string_in)))
; Define the variable with dimensions zid, yid, xid:
id = NCDF_VARDEF(ncdfid, 'strings', [zid,yid,xid], /CHAR)
; Put the file into define mode:
NCDF_CONTROL, ncdfid, /ENDEF
; Write the string variable. The array will be stored as bytes
; in the file:
NCDF_VARPUT, ncdfid, id, string_in
; Read the byte array back out:
NCDF_VARGET, ncdfid, id, byte_out
NCDF_CLOSE, ncdfid ; Close the file.
HELP, byte_out
; IDL reports that BYTE_OUT is a (16, 10, 10) BYTE array.
PRINT, STRING(byte_out(*,0,0))
; Taking the STRING of the first "row" of byte_out returns the
; first element of our original array, "Long Test String".
; Convert the entire byte array back into strings:
string_new = STRING(byte_out)
; The new string array has the same dimensions and values as
; our original string, string_in.
HELP, string_new
; This statement compares the two arrays and prints "Success!" if
; they are equal, and they are:
IF TOTAL(string_in NE string_new) EQ 0 THEN PRINT, 'Success!'
```

Alphabetical Listing of NCDF Routines

NCDF_ATTCOPY

NCDF_ATTDEL

NCDF_ATTGET

NCDF_ATTINQ

NCDF_ATTNAME

NCDF_ATTPUT

NCDF_ATTRENAME

NCDF_CLOSE

NCDF_CONTROL

NCDF_CREATE

NCDF_DIMDEF

NCDF_DIMID

NCDF_DIMINQ

NCDF_DIMRENAME

NCDF_EXISTS

NCDF_INQUIRE

NCDF_OPEN

NCDF_VARDEF

NCDF_VARGET

NCDF_VARID

NCDF_VARINQ

NCDF_VARPUT

NCDF_VARRENAME

NCDF_ATTCOPY

The NCDF_ATTCOPY function copies an attribute from one netCDF file to another. Note that *Incdf* and *Outcdf* can be the same netCDF ID.

Syntax

Result = NCDF_ATTCOPY(*Incdf* [, *Invar*] , *Name*, *Outcdf* [, *Outvar*]
[, /IN_GLOBAL] [, /OUT_GLOBAL])

Return Value

NCDF_ATTCOPY returns the attribute number of the copied attribute in the new file, or -1 if the copy was not successful.

Arguments

Incdf

The netCDF ID, returned from a previous call to NCDF_OPEN or NCDF_CREATE.

Invar

The netCDF variable ID to be read, returned from a previous call to NCDF_VARDEF or NCDF_VARID, or the name of the variable. If the IN_GLOBAL keyword is set, this argument must be omitted.

Name

A scalar string containing the name of the attribute to be copied.

Outcdf

The netCDF ID of a netCDF file opened for writing, returned from a previous call to NCDF_OPEN or NCDF_CREATE.

Outvar

The netCDF variable ID to be written, returned from a previous call to NCDF_VARDEF or NCDF_VARID, or the name of the variable. If the OUT_GLOBAL keyword is set, this argument must be omitted.

Keywords

IN_GLOBAL

Set this keyword to read a global attribute.

OUT_GLOBAL

Set this keyword to create a global attribute.

Examples

See example from "NCDF_ATTINQ" on page 836.

Version History

Pre 4.0	Introduced

NCDF_ATTDEL

The NCDF_ATTDEL procedure deletes an attribute from a netCDF file.

Syntax

NCDF_ATTDEL, *Cdfid* [, *Varid*] , *Name* [, /GLOBAL]

Arguments

Cdfid

The netCDF ID, returned from a previous call to NCDF_OPEN or NCDF_CREATE.

Varid

The netCDF variable ID, returned from a previous call to NCDF_VARDEF or NCDF_VARID, or the name of the variable. If the GLOBAL keyword is used, this argument must be omitted.

Name

A scalar string containing the name of the attribute to be deleted.

Keywords

GLOBAL

Set this keyword to delete a global variable.

Examples

```
; Open file test.nc for writing:
id = NCDF_OPEN('test.nc', /WRITE)
; Delete global attribute TITLE from the file:
NCDF_ATTDEL, id, 'TITLE', /GLOBAL
NCDF_CLOSE, id ; Close the file.
```

Version History

Pre 4.0	Introduced

See Also

NCDF_ATTNAME, NCDF_ATTPUT

NCDF_ATTGET

The NCDF_ATTGET procedure retrieves the value of an attribute from a netCDF file.

Syntax

NCDF_ATTGET, *Cdfid* [, *Varid*] , *Name*, *Value* [, /GLOBAL]

Arguments

Cdfid

The netCDF ID, returned from a previous call to NCDF_OPEN or NCDF_CREATE.

Varid

The netCDF variable ID, returned from a previous call to NCDF_VARDEF or NCDF_VARID, or the name of the variable. If the GLOBAL keyword is used, this argument must be omitted.

Name

A scalar string containing the attribute name.

Value

A named variable in which the attribute's value is returned. NCDF_ATTGET sets *Value*'s size and data type appropriately.

Keywords

GLOBAL

Set this keyword to retrieve the value of a global attribute.

Examples

For an example using this routine, see the documentation for NCDF_ATTINQ.

Version History

Pre 4.0	Introduced

See Also

NCDF_ATTINQ, NCDF_ATTNAME, NCDF_ATTPUT

NCDF_ATTINQ

The NCDF_ATTINQ function returns a structure that contains information about a netCDF attribute. This structure, described below, has the form:

```
{ DATATYPE:'', LENGTH:0L }
```

Syntax

Result = NCDF_ATTINQ(*Cdfid* [, *Varid*] , *Name* [, /GLOBAL])

Return Value

The structure returned by this function contains the following tags:

Tag	Description
DataType	A string describing the data type of the variable. The string will be one of the following: BYTE, CHAR, INT, LONG, FLOAT, or DOUBLE.
Length	The number of values stored in the attribute. If the attribute is a string, the number of values indicates one more character than the string length to include the terminating null character. This is the NetCDF convention, as demonstrated in the following example.

Table 6-1: NCDF_ATTINQ Structure Tags

Arguments

Cdfid

The netCDF ID, returned from a previous call to NCDF_OPEN or NCDF_CREATE.

Varid

The netCDF variable ID, returned from a previous call to NCDF_VARDEF or NCDF_VARID, or the name of the variable. If the GLOBAL keyword is set, this argument must be omitted.

Name

A scalar string containing the name of the attribute for which information is to be returned.

Keywords

GLOBAL

Set this keyword to inquire about a global variable. If this keyword is set, the *Varid* argument must be omitted.

Examples

```
id = NCDF_CREATE('test.nc', /CLOBBER ; Open a new netCDF file.
id2 = NCDF_CREATE('test2.nc', /CLOBBER ; Open a second file.
; Create two global attributes TITLE and DATE:
NCDF_ATTPUT, id, /GLOBAL, 'TITLE', 'MY TITLE'
NCDF_ATTPUT, id, /GLOBAL, 'DAY', 'July 1,1996'
; Suppose we wanted to use DATE instead of DAY. We could use
; ATTRENAME to rename the attribute:
NCDF_ATTRENAME, id, 'DAY', 'DATE', /GLOBAL
; Next, copy both attributes into a duplicate file:
result = NCDF_ATTCOPY(id, 'TITLE', id2, /IN_GLOBAL, /OUT_GLOBAL)
result2 = NCDF_ATTCOPY(id, 'DATE', id2, /IN_GLOBAL, /OUT_GLOBAL)
; Put the file into data mode:
NCDF_CONTROL, id, /ENDEF
; Get the second attribute’s name:
name = NCDF_ATTNAME(id, /GLOBAL, 1)
; Retrieve the date:
NCDF_ATTGET, id, /GLOBAL, name, date
; Get info about the attribute:
result = NCDF_ATTINQ(id, /GLOBAL, name)
HELP, name, date, result, /STRUCTURE
PRINT, date
PRINT, STRING(date)
NCDF_DELETE, id ; Close the netCDF files.
NCDF_DELETE, id2
```

IDL Output

```
NAME            STRING    = 'DATE'
DATE            BYTE      = Array(12)
** Structure <400dac30>, 2 tags, length=12, refs=1:
   DATATYPE        STRING    'BYTE'
   LENGTH          LONG                12
```

Note the length includes the NCDF standard NULL terminator

```
74 117 108 121  32  49  44  49  57  57  54   0

July 1,1996
```

Version History

Pre 4.0	Introduced

See Also

NCDF_ATTDEL, NCDF_ATTGET, NCDF_ATTNAME, NCDF_ATTPUT

NCDF_ATTNAME

The NCDF_ATTNAME function returns the name of an attribute in a netCDF file given its ID.

Syntax

Result = NCDF_ATTNAME(*Cdfid* [, *Varid*] , *Attnum* [, /GLOBAL])

Return Value

Returns the specified attribute’s name or the NULL string ("") if there is no such attribute.

Arguments

Cdfid

The netCDF ID, returned from a previous call to NCDF_OPEN or NCDF_CREATE.

Varid

The netCDF variable ID, returned from a previous call to NCDF_VARDEF or NCDF_VARID, or the name of the variable. If the GLOBAL keyword is set, this argument must be omitted.

Attnum

An expression containing the number of the desired attribute. The attributes for each variable are numbered from 0 to the number-of-attributes minus 1. Note that the number of attributes can be found using NCDF_VARINQ or NCDF_INQUIRE (to find the number of global variables).

Keywords

GLOBAL

Set this keyword to return the name of one of the global attributes.

Version History

Pre 4.0	Introduced

See Also

NCDF_ATTINQ

NCDF_ATTPUT

The NCDF_ATTPUT procedure creates an attribute in a netCDF file. If the attribute is new, or if the space required to store the attribute is greater than before, the netCDF file must be in *define* mode.

Syntax

NCDF_ATTPUT, *Cdfid* [, *Varid*] , *Name* , *Value* [, /GLOBAL] [, LENGTH=*value*]
[, /BYTE | , /CHAR | , /DOUBLE | , /FLOAT | , /LONG | , /SHORT]

Arguments

Cdfid

The netCDF ID, returned from a previous call to NCDF_OPEN or NCDF_CREATE.

Varid

The netCDF variable ID, returned from a previous call to NCDF_VARDEF or NCDF_VARID, or the name of the variable. If the GLOBAL keyword is set, this argument must be omitted.

Name

A scalar string containing the attribute name.

Warning

The *Name* string may contain only alphanumeric characters and the **-** , **_** , and **.** characters.

Value

An expression containing the data to be written. Although this procedure checks that there are a sufficient number of bytes of data, the data type is not checked or altered.

Keywords

GLOBAL

Set this keyword to create a global attribute.

LENGTH

Use this keyword to override the default length (the whole value). Set this keyword to a value less than or equal to the number of elements in *Value*. For example:

```
ATTR_ID = NCDF_ATTPUT(CDFID, VARID, 'Attr1', $
   INDGEN(10), LENGTH=5
```

writes Attr1 as [0,1,2,3,4].

The following keywords specify a non-default data type for the variable. By default, NCDF_ATTPUT chooses one based upon the type of data. If a data type flag is specified, the data supplied in *Value* is converted to that data type before being written to the file. Only one of these keywords can be used in a single call to NCDF_ATTPUT.

BYTE

Set this keyword to indicate that the data is composed of bytes.

CHAR

Set this keyword to indicate that the data is composed of bytes (assumed to be ASCII).

DOUBLE

Set this keyword to indicate that the data is composed of 8-byte floating point numbers (doubles).

FLOAT

Set this keyword to indicate that the data is composed of 4-byte floating point numbers (floats).

LONG

Set this keyword to indicate that the data is composed of 4-byte integers (longs).

SHORT

Set this keyword to indicate that the data is composed of 2-byte integers.

Examples

```
NCDF_ATTPUT, cdfid, /GLOBAL, "Title", "My Favorite Data File"
NCDF_ATTPUT, cdfid, "data", "scale_factor", 12.5D
```

Version History

Pre 4.0	Introduced

See Also

NCDF_ATTINQ

NCDF_ATTRENAME

The NCDF_ATTRENAME procedure renames an attribute in a netCDF file.

Syntax

NCDF_ATTRENAME, *Cdfid* [, *Varid*] *Oldname*, *Newname* [, /GLOBAL]

Arguments

Cdfid

The netCDF ID, returned from a previous call to NCDF_OPEN or NCDF_CREATE.

Varid

The netCDF variable ID, returned from a previous call to NCDF_VARDEF or NCDF_VARID, or the name of the variable. If the GLOBAL keyword is set, this argument must be omitted.

OldName

A scalar string containing the attribute's current name.

NewName

A scalar string containing the attribute's new name.

Keywords

GLOBAL

Set this keyword to rename a global attribute.

Version History

Pre 4.0	Introduced

See Also

NCDF_ATTINQ

NCDF_CLOSE

The NCDF_CLOSE procedure closes an open netCDF file. If a writable netCDF file is not closed before exiting IDL, the disk copy of the netCDF file may not reflect recent data changes or new definitions.

Syntax

NCDF_CLOSE, *Cdfid*

Arguments

Cdfid

The netCDF ID of the file to be closed, returned from a previous call to NCDF_OPEN or NCDF_CREATE.

Keywords

None

Version History

Pre 4.0	Introduced

See Also

NCDF_ATTINQ

NCDF_CONTROL

The NCDF_CONTROL procedure performs miscellaneous netCDF operations.

Different options are controlled by keywords. Only one keyword can be specified in any call to NCDF_CONTROL, unless the OLDFILL keyword is specified.

Syntax

NCDF_CONTROL, *Cdfid* [, /ABORT] [, /ENDEF] [, /FILL | , /NOFILL]
[, /NOVERBOSE | , /VERBOSE] [, OLDFILL=*variable*] [, /REDEF] [, /SYNC]

Arguments

Cdfid

The netCDF ID, returned from a previous call to NCDF_OPEN or NCDF_CREATE.

Keywords

ABORT

Set this keyword to close a netCDF file that is not in define mode. If the file is being created and is still in define mode, the file is deleted. If define mode was entered by a call to NCDF_CONTROL with the REDEF keyword, the netCDF file is restored to its state before definition mode was entered, and the file is closed.

ENDEF

Set this keyword to take an open netCDF file out of define mode (and into data mode).

FILL

Set this keyword so that data in the netCDF file is pre-filled with default fill values. The default values (which cannot be changed) are:

Data Type	Fill Value
BYTE	0
CHAR	0
SHORT	-32767
LONG	-2147483647
FLOAT	9.96921E+36
DOUBLE	9.96921E+36

Table 6-2: Default Fill Values for netCDF Files

NOFILL

Set this keyword so that data in the netCDF file is not pre-filled. This option saves time when it is certain that variable values will be written before a read is attempted.

NOVERBOSE

Set this keyword to suppress the printing of netCDF error messages. *Cdfid* is required but not used.

OLDFILL

This keyword specifies a named variable in which the previous fill value is returned. This keyword can only be used in combination with the FILL or NOFILL keywords. For example:

```
NCDF_CONTROL, id, FILL=1, OLDFILL=previous_fill
```

REDEF

Set this keyword to put an open netCDF file into define mode.

SYNC

Set this keyword to update the disk copy of a netCDF file that is open for writing. The netCDF file must be in data mode. A netCDF file in define mode will be updated only when NCDF_CONTROL is called with the ENDEF keyword.

VERBOSE

Set this keyword to cause netCDF error messages to be printed. *Cdfid* is required but not used. For example:

```
NCDF_CONTROL, 0, /VERBOSE
```

is a valid command even if 0 is not a valid NetCDF file ID.

Examples

See the examples under NCDF_ATTINQ and NCDF_VARPUT.

Version History

Pre 4.0	Introduced

See Also

NCDF_CLOSE, NCDF_CREATE, NCDF_OPEN

NCDF_CREATE

The NCDF_CREATE function creates a new netCDF file.

Syntax

Result = NCDF_CREATE(*Filename* [, /CLOBBER | , /NOCLOBBER])

Return Value

If successful, the netCDF ID for the file is returned. The newly-created netCDF file is automatically placed into define mode. If you do not have write permission to create the specified Filename, NCDF_CREATE returns an error message instead of a netCDF file ID.

Arguments

Filename

A scalar string containing the name of the file to be created

Keywords

CLOBBER

Set this keyword to erase the existing file (if the file already exists) before creating the new version.

NOCLOBBER

Set this keyword to create a new netCDF file only if the specified file does not already exist. This is the default.

Examples

```
; Open a new NetCDF File and destroy test.nc if it already exists:
id = NCDF_CREATE('test.nc',/CLOBBER)

id2 = NCDF_CREATE('test.nc', /NOCLOBBER)
```

This attempt to create a new version of the file `test.nc` produces the following error because the NOCLOBBER keyword was set:

```
nccreate: filename "test.nc": File exists
% NCDF_CREATE: Operation failed
% Execution halted at $MAIN$   (NCDF_CREATE).
```

Version History

Pre 4.0	Introduced

See Also

NCDF_CLOSE, NCDF_CONTROL, NCDF_OPEN

NCDF_DIMDEF

The NCDF_DIMDEF function defines a dimension in a netCDF file given its name and size.

Syntax

Result = NCDF_DIMDEF(*Cdfid*, *DimName*, *Size* [, /UNLIMITED])

Return Value

If successful, the dimension ID is returned.

Arguments

Cdfid

The netCDF ID, returned from a previous call to NCDF_OPEN or NCDF_CREATE.

DimName

A scalar string containing the name of the dimension being defined.

Size

The size of the dimension. *Size* can be any scalar expression. If the UNLIMITED keyword is used, the *Size* parameter should be omitted.

Keywords

UNLIMITED

Set this keyword to create a dimension of unlimited size. Note that only one dimension in a netCDF file can be unlimited.

Examples

See NCDF_VARPUT.

Version History

Pre 4.0	Introduced

NCDF_DIMID

The NCDF_DIMID function returns the ID of a netCDF dimension, given the name of the dimension.

Syntax

Result = NCDF_DIMID(*Cdfid*, *DimName*)

Return Value

Return the dimension ID or -1 if the dimension does not exist.

Arguments

Cdfid

The netCDF ID, returned from a previous call to NCDF_OPEN or NCDF_CREATE.

DimName

A scalar string containing the dimension name.

Keywords

None

Examples

See NCDF_VARPUT.

Version History

Pre 4.0	Introduced

NCDF_DIMINQ

The NCDF_DIMINQ procedure retrieves the name and size of a dimension in a netCDF file, given its ID. The size for the unlimited dimension, if any, is the maximum value used so far in writing data for that dimension.

Syntax

NCDF_DIMINQ, *Cdfid*, *Dimid*, *Name*, *Size*

Arguments

Cdfid

The netCDF ID, returned from a previous call to NCDF_OPEN or NCDF_CREATE.

Dimid

The netCDF dimension ID, returned from a previous call to NCDF_DIMID or NCDF_DIMDEF, or an indexed number from 0 to NDIMS-1 that indexes the desired dimension. The first dimension has a DIMID of 0, the second dimension has a DIMID of 1, and so on.

Name

A named variable in which the dimension name is returned (a scalar string).

Size

A named variable in which the size of the dimension is returned (a scalar longword integer)

Keywords

None

Examples

See NCDF_VARPUT.

Version History

Pre 4.0	Introduced

NCDF_DIMRENAME

The NCDF_DIMRENAME procedure renames an existing dimension in a netCDF file which has been opened for writing. If the new name is longer than the old name, the netCDF file must be in define mode. You cannot rename one dimension to have the same name as another dimension.

Syntax

NCDF_DIMRENAME, *Cdfid*, *Dimid*, *NewName*

Arguments

Cdfid

The netCDF ID, returned from a previous call to NCDF_OPEN or NCDF_CREATE.

Dimid

The netCDF dimension ID, returned from a previous call to NCDF_DIMID or NCDF_DIMDEF, or the name of the dimension.

NewName

A scalar string containing the new name for the dimension.

Keywords

None

Examples

See NCDF_VARPUT.

Version History

Pre 4.0	Introduced

NCDF_EXISTS

The NCDF_EXISTS function returns true if the Network Common Data Format (netCDF) scientific data format library is supported on the current IDL platform.

This routine is written in the IDL language. Its source code can be found in the file `ncdf_exists.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = NCDF_EXISTS( )

Return Value

Returns true if the library is supported.

Arguments

None

Keywords

None

Examples

The following IDL command prints an error message if the NetCDF library is not available:

```
IF NCDF_EXISTS() EQ 0 THEN PRINT, 'NCDF not supported.'
```

Version History

Pre 4.0	Introduced

NCDF_INQUIRE

The NCDF_INQUIRE function returns a structure that contains information about an open netCDF file. This structure of the form:

```
{ NDIMS:0L, NVARS:0L, NGATTS:0L, RECDIM:0L }
```

The structure tags are described below.

Syntax

Result = NCDF_INQUIRE(*Cdfid*)

Return Value

The returned structure contains the following tags:

Tag	Description
Ndims	The number of dimensions defined for this netCDF file.
Nvars	The number of variables defined for this netCDF file.
Ngatts	The number of global attributes defined for this netCDF file.
RecDim	The ID of the unlimited dimension, if there is one, for this netCDF file. If there is no unlimited dimension, RecDim is set to -1.

Table 6-3: NCDF_INQUIRE Structure Tags

Arguments

Cdfid

The netCDF ID, returned from a previous call to NCDF_OPEN or NCDF_CREATE.

Keywords

None

Examples

See NCDF_VARDEF.

Version History

Pre 4.0	Introduced

NCDF_OPEN

The NCDF_OPEN function opens an existing netCDF file.

Syntax

Result = NCDF_OPEN(*Filename* [, /NOWRITE | , /WRITE])

Return Value

If successful, the netCDF ID for the file is returned.

Arguments

Filename

A scalar string containing the name of the file to be opened.

Keywords

NOWRITE

Set this keyword to open an existing netCDF file as read only. This is the default.

WRITE

Set this keyword to open an existing netCDF file for both writing and reading.

Version History

Pre 4.0	Introduced

See Also

NCDF_ATTINQ

NCDF_VARDEF

The NCDF_VARDEF function adds a new variable to an open netCDF file in define mode.

Syntax

Result = NCDF_VARDEF(*Cdfid*, *Name* [, *Dim*] [, /BYTE | , /CHAR | , /DOUBLE | , /FLOAT | , /LONG | , /SHORT])

Return Value

If successful, the variable ID is returned. If a new variable cannot be defined, NCDF_VARDEF returns -1.

Arguments

Cdfid

The netCDF ID, returned from a previous call to NCDF_OPEN or NCDF_CREATE.

Name

A scalar string containing the variable name.

Dim

An optional vector containing the dimension IDs corresponding to the variable dimensions. If the ID of the unlimited dimension is included, it must be the rightmost element in the array. If *Dim* is omitted, the variable is assumed to be a scalar.

Keywords

The following keywords specify the data type for the variable. Only one of these keywords can be used. If no data type keyword is specified, FLOAT is used by default.

BYTE

Set this keyword to indicate that the data is composed of bytes.

CHAR

Set this keyword to indicate that the data is composed of bytes (assumed to be ASCII).

DOUBLE

Set this keyword to indicate that the data is composed of double-precision floating-point numbers.

FLOAT

Set this keyword to indicate that the data is composed of floating-point numbers.

LONG

Set this keyword to indicate that the data is composed of longword integers.

SHORT

Set this keyword to indicate that the data is composed of 2-byte integers.

Examples

```
id = NCDF_CREATE('test.nc', /CLOBBER) ; Create the netCDF file.
NCDF_ATTPUT, id, 'TITLE', 'Incredibly Important Data', /GLOBAL
NCDF_ATTPUT, id, 'GALAXY', 'Milky Way', /GLOBAL
NCDF_ATTPUT, id, 'PLANET', 'Earth', /GLOBAL
xid = NCDF_DIMDEF(id, 'x', 100) ; Define the X dimension.
yid = NCDF_DIMDEF(id, 'y', 200) ; Define the Y dimension.
zid = NCDF_DIMDEF(id, 'z', /UNLIMITED) ; Define the Z dimension.
vid0 = NCDF_VARDEF(id, 'image0', [yid, xid], /FLOAT)
vid1 = NCDF_VARDEF(id, 'image1', [yid, xid], /FLOAT)
; Rename image0 to dist_image:
dist_id = NCDF_VARID(id, 'image0')
NCDF_VARRENAME, id, vid0, 'dist_image'
NCDF_ATTPUT, id, vid, 'TITLE', 'DIST_IMAGE'
NCDF_CONTROL, id, /ENDEF ; Put the file into data mode.
image = CONGRID(DIST(200), 200, 100)
NCDF_VARPUT, id, vid, image
INQ_VID = NCDF_VARINQ(id, 'dist_image')
HELP, INQ_VID, /STRUCTURE
file_inq = NCDF_INQUIRE(id)
HELP, file_inq, /STRUCTURE
NCDF_CLOSE, id ; Close the NetCDF file.
```

IDL Output

```
** Structure <400ec678>, 5 tags, length=32, refs=1:
   NAME            STRING    'dist_image'
   DATATYPE        STRING    'FLOAT'
   NDIMS           LONG                 2
   NATTS           LONG                 1
   DIM             LONG      Array(2)
** Structure <400ebdf8>, 4 tags, length=16, refs=1:
   NDIMS           LONG                 3
   NVARS           LONG                 2
   NGATTS          LONG                 3
   RECDIM          LONG                 2
```

Version History

Pre 4.0	Introduced

NCDF_VARGET

The NCDF_VARGET procedure retrieves a hyperslab of values from a netCDF variable. The netCDF file must be in *data* mode to use this procedure.

Syntax

NCDF_VARGET, *Cdfid*, *Varid*, *Value* [, COUNT=*vector*] [, OFFSET=*vector*]
[, STRIDE=*vector*]

Arguments

Cdfid

The netCDF ID, returned from a previous call to NCDF_OPEN or NCDF_CREATE.

Varid

The netCDF variable ID, returned from a previous call to NCDF_VARDEF or NCDF_VARID, or the name of the variable.

Value

A named variable in which the values are returned. NCDF_VARGET sets *Value*'s size and data type as appropriate.

Keywords

COUNT

An optional vector containing the counts to be used in reading *Value*. COUNT is a 1-based vector with an element for each dimension of the data to be written.The default matches the size of the variable so that all data is written out.

OFFSET

An optional vector containing the starting position for the read. The default start position is [0, 0, ...].

STRIDE

An optional vector containing the strides, or sampling intervals, between accessed values of the netCDF variable. The default stride vector is that for a contiguous read, [1, 1, ...].

Examples

Suppose that a 230 by 230 image is saved in the netCDF file `dave.nc`. The following commands extract both the full image and a 70x70 sub-image starting at [80,20] sampling every other X pixel and every third Y pixel:

```
; A variable that contains the offset for the sub-image:
offset = [80, 20]
; The dimensions of the sub-image:
count = [70, 70]
; Create a variable to be used as a value for the STRIDE keyword.
; Every other X element and every third Y element will be sampled:
stride = [2, 3]
; Open the NetCDF file:
id = NCDF_OPEN('dave.nc')
; Get the variable ID for the image:
image = NCDF_VARID(id, 'image')
; Get the full image:
NCDF_VARGET, id, image, fullimage
; Extract the sub-sampled image:
NCDF_VARGET, id, image, subimage, $
   COUNT=count, STRIDE=stride, OFFSET=offset
; Close the NetCDF file:
NCDF_CLOSE, id
```

Version History

Pre 4.0	Introduced

See Also

NCDF_VARGET1, NCDF_VARID, NCDF_VARINQ, NCDF_VARPUT

NCDF_VARGET1

The NCDF_VARGET1 procedure retrieves one element from a netCDF variable. The netCDF file must be in *data* mode to use this procedure.

Syntax

NCDF_VARGET1, *Cdfid*, *Varid*, *Value* [, OFFSET=*vector*]

Arguments

Cdfid

The netCDF ID, returned from a previous call to NCDF_OPEN or NCDF_CREATE.

Varid

The netCDF variable ID, returned from a previous call to NCDF_VARDEF or NCDF_VARID, or the name of the variable.

Value

A named variable in which the value of the variable is returned. NCDF_VARGET1 sets *Value*'s size and data type as appropriate.

Keywords

OFFSET

A vector containing the starting position of the read. The default starting position is [0, 0, ...].

Examples

Suppose that the file `dave.nc` contains an image saved with the netCDF variable name "dave". The following commands extract the value of a single pixel from the image:

```
; The location of the single element (pixel) whose value we will
; retrieve:
offset = [180,190]
; Open the netCDF file:
id = NCDF_OPEN('dave.nc')
```

```
; Get the variable ID for variable "dave":
varid = NCDF_VARID(id, 'dave')
; Extract the element and return the value in the variable
; single_pixel:
NCDF_VARGET1, id, varid, single_pixel, OFFSET=offset
; Close the netCDF file:
NCDF_CLOSE, id
```

Version History

Pre 4.0	Introduced

See Also

NCDF_VARGET, NCDF_VARID, NCDF_VARINQ, NCDF_VARPUT

NCDF_VARID

The NCDF_VARID function returns the ID of a netCDF variable.

Syntax

Result = NCDF_VARID(*Cdfid*, *Name*)

Return Value

This function returns the variable ID or returns -1 if the variable does not exist.

Arguments

Cdfid

The netCDF ID, returned from a previous call to NCDF_OPEN or NCDF_CREATE.

Name

A scalar string containing the variable name.

Keywords

None

Examples

See NCDF_VARDEF.

Version History

Pre 4.0	Introduced

NCDF_VARINQ

The NCDF_VARINQ function returns a structure that contains information about a netCDF variable, given its ID. This structure has the form:

```
{ NAME:"", DATATYPE:"", NDIMS:0L, NATTS:0L, DIM:LONARR(NDIMS) }
```

This structure is described below.

Syntax

Result = NCDF_VARINQ(*Cdfid*, *Varid*)

Return Value

The returned structure contains the following tags:

Tag	Description
Name	The name of the variable.
DataType	A string describing the data type of the variable. The string will be one of the following: 'BYTE', 'CHAR', 'INT', 'LONG', 'FLOAT', or 'DOUBLE'.
Ndims	The number of dimensions.
Natts	The number of attributes assigned to this variable.
Dim	A vector of the dimension IDs for the variable dimensions.

Table 6-4: NCDF_VARINQ Structure Tags

Arguments

Cdfid

The netCDF ID, returned from a previous call to NCDF_OPEN or NCDF_CREATE.

Varid

The netCDF variable ID, returned from a previous call to NCDF_VARDEF or NCDF_VARID, or the name of the variable.

Examples

See NCDF_VARDEF.

Version History

Pre 4.0	Introduced

NCDF_VARPUT

The NCDF_VARPUT procedure writes a hyperslab of values to a netCDF variable. The netCDF file must be in *data* mode to use this procedure.

Syntax

NCDF_VARPUT, *Cdfid, Varid, Value* [, COUNT=*vector*] [, OFFSET=*vector*]
[, STRIDE=*vector*]

Arguments

Cdfid

The netCDF ID, returned from a previous call to NCDF_OPEN or NCDF_CREATE.

Varid

The netCDF variable ID, returned from a previous call to NCDF_VARDEF or NCDF_VARID, or the name of the variable.

Value

Data values to be written to the netCDF file. If the data type of *Value* does not match that of the netCDF variable, it is converted to the correct data type before writing. *Value* must have a dimensionality less than or equal to that of the variable being written.

Keywords

COUNT

An optional vector containing the counts to be used in writing *Value*. COUNT is a 1-based vector with an element for each dimension of the data to be written. Note that counts do not have to match the dimensions of *Value*. The default count vector is the dimensionality of *Value*.

OFFSET

An optional vector containing the starting position to write. The default start position is [0, 0, ...].

STRIDE

An optional vector containing the strides, or writing intervals, between written values of the netCDF variable. The default stride vector is that for a contiguous write, [1, 1, ...].

Examples

Suppose that you wish to create a 100x100 byte (0 & 1) checker board:

```
; Create offsets for even and odd rows:
offset_even = [0,0] & offset_odd = [1,1]
; Create count and stride values:
count = [50,50] & stride = [2,2]
; Make the "black" spaces of the checker board:
black = BYTARR(50,50) > 1B
; Create the netCDF file:
id = NCDF_CREATE('checker.nc', /CLOBBER)
; Fill the file with BYTE zeros:
NCDF_CONTROL, id, /FILL
; Define the X dimension:
xid = NCDF_DIMDEF(id, 'x', 100)
; Define the Y dimension:
yid = NCDF_DIMDEF(id, 'y', 100)
; Define the Z dimension, UNLIMITED:
zid = NCDF_DIMDEF(id, 'yy', /UNLIMITED)
; Define a variable with the name "board":
vid = NCDF_VARDEF(id, 'board', [yid, xid], /BYTE)
; Rename 'yy' to 'z' as the zid dimension name:
NCDF_DIMRENAME, id, zid, 'z'
; Put the file into data mode:
NCDF_CONTROL, id, /ENDEF
; Use NCDF_DIMID and NCDF_DIMINQ to verify the name and size
; of the zid dimension:
check_id = NCDF_DIMID(id,'z')
NCDF_DIMINQ, id, check_id, dim_name, dim_size
HELP, check_id, dim_name, dim_size
```

IDL prints:

```
CHECK_ID     LONG            =   2
DIM_NAME     STRING          = 'z'
DIM_SIZE     LONG            =   0
```

Note that the DIM_SIZE is 0 because no records have been written yet for this dimension.

```
NCDF_VARPUT, id, vid, black, $
   COUNT=count, STRIDE=stride, OFFSET=offset_even
NCDF_VARPUT, id, vid, black, $
   COUNT=count, STRIDE=stride, OFFSET=offset_odd
; Get the full image:
NCDF_VARGET, id, vid, output
; Create a window for displaying the image:
WINDOW, XSIZE=100, YSIZE=100
; Display the image:
TVSCL, output
; Make stride larger than possible:
stride = [2,3]
; As an experiment, attempt to write to an array larger than
; the one we previously allocated with NCDF_VARDEF:
NCDF_VARPUT, id, vid, black, $
   COUNT=count, STRIDE=stride, OFFSET=offset_odd
```

IDL prints:

```
% NCDF_VARPUT: Requested write is larger than the available data
area.
```

You will need to change the OFFSET/COUNT/STRIDE, or redefine the variable dimensions. You attempted to access 150 elements in a 100 array.

```
NCDF_CLOSE, id ; Close the netCDF file.
```

Version History

Pre 4.0	Introduced

See Also

NCDF_VARGET, NCDF_VARGET1, NCDF_VARID, NCDF_VARINQ

NCDF_VARRENAME

The NCDF_VARRENAME procedure renames a netCDF variable.

Syntax

NCDF_VARRENAME, *Cdfid*, *Varid*, *Name*

Arguments

Cdfid

The netCDF ID, returned from a previous call to NCDF_OPEN or NCDF_CREATE.

Varid

The netCDF variable ID, returned from a previous call to NCDF_VARDEF or NCDF_VARID, or the name of the variable.

Name

A scalar string containing the new name for the variable.

Keywords

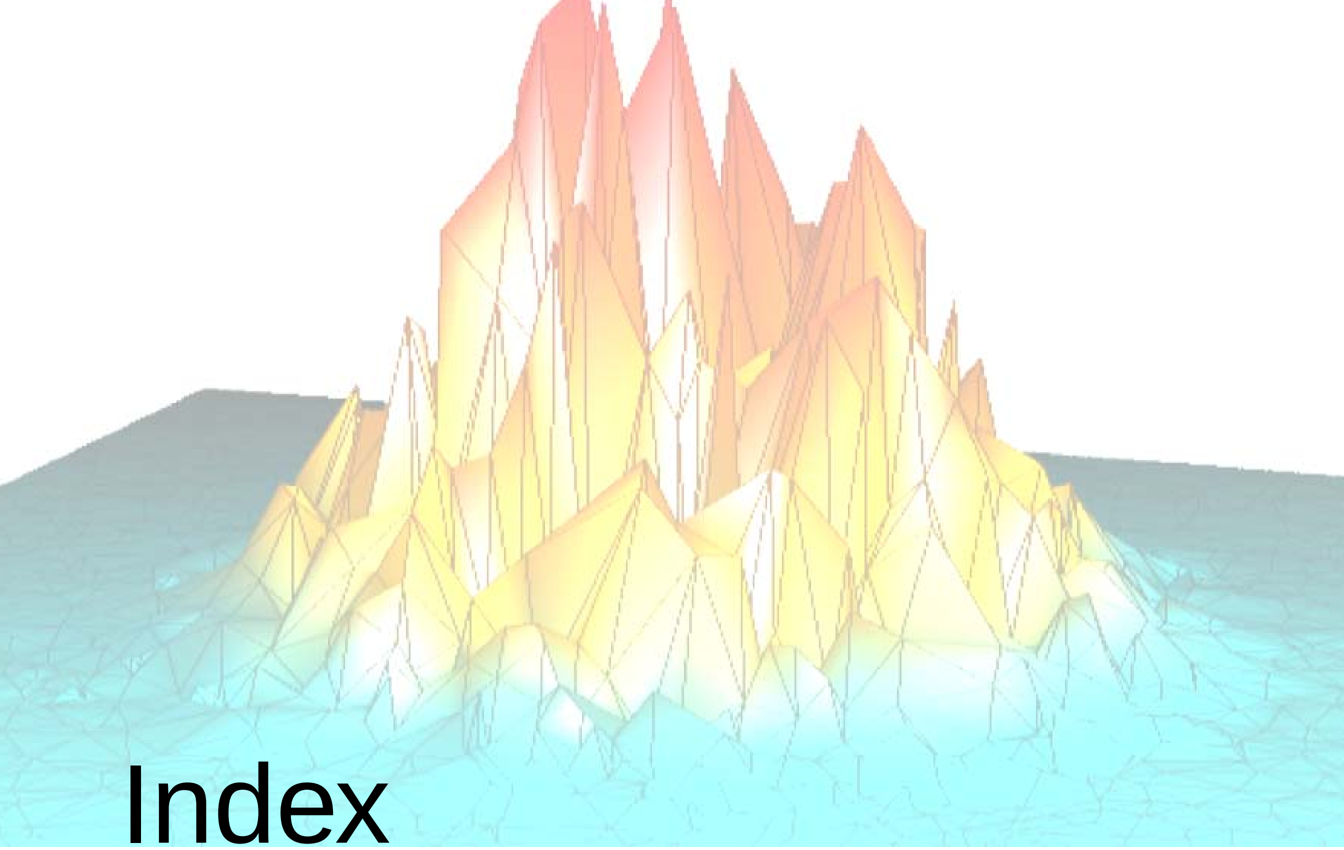
None

Examples

See NCDF_VARDEF.

Version History

Pre 4.0	Introduced



Index