* Function Specifications

# Overview

This appendix presents the detailed specifications of selected individual routines of the HDF low level interface. Several low level routines are documented in the HDF Reference Manual and all are documented in the distributed source code.

The terms IN: and OUT: indicate whether parameters are input or output parameters; in some cases, a parameter may be both. In the following specifications, these terms should be interpreted as follows:

IN: Value as input parameter

OUT: Value as output parameter

## Opening and Closing Files

Hopen

int32 Hopen(char \*path, int access, int16 ndds)

*path* IN: Complete path and name of the file to be opened

*access* IN: DFACC\_READ, DFACC\_CREATE, or DFACC\_WRITE

*ndds* IN: Number of DDs in a block if this file needs to be created

Purpose Provides an access path to an HDF file and reads all of the DD blocks in the file into primary memory.

Return value Returns file ID if successful and FAIL (-1) otherwise.

Description Opens an HDF file.

The following events occur on successful exit:

* File\_rec members are filled in. (File\_rec is an internal HDF structure containing information about the opened file.)
* The requested file is opened with the relevant permission.
* Information about DDs is set up in memory.
* The file headers and initial information are set up for new files.

Access privilege codes  
HDF provides several constants for use as access privilege codes as listed below. Note that these constants are not bit-flags and should not be ORed together to combine access modes. Doing so may cause odd behavior and, in some cases, loss of data:

Recommended tags:

DFACC\_READ Open for read only. If file does not exist, error.

DFACC\_WRITE Open for read/write. If file does not exist, create it.

DFACC\_CREATE Force creation. If file exists, delete it, then open a new file for read/write (in the spirit of the UNIX system command clobber).

Obsolete tags:

DFACC\_ALL Same as DFACC\_WRITE (obsolete but still supported).

DFACC\_RDWR Same as DFACC\_WRITE (obsolete but still supported).

Hclose

intn Hclose(int32 id)

*id* IN: The identifier of the file to be closed

Purpose Closes the access path to the file.

Return value Returns SUCCEED (0) if successful and FAIL (-1) otherwise.

Description *id* is first validated. If valid, the function closes the access path to the file.

If there are still access elements attached to the file, the error DFE\_OPENAID is pushed onto the error stack and the file is not closed. This is a fairly common error when developing new interfaces. See the discussion of Hendaccess below for debugging hints.

## Locating Elements for Access and Getting Information

Hstartread

int32 Hstartread(int32 file\_id, uint16 tag, uint16 ref)

*file\_id* IN: ID of file to attach access element to

*tag* IN: Tag to search for

*ref* IN: Reference number to search for

Purpose Locates an existing data element with matching tag/ref and returns an access ID for reading it.

Return value Returns access element ID if successful and FAIL (-1) otherwise.

DescriptionSearches the DDs for a particular tag/ref combination. If the search is successful, an access element is created, attached to the file, and positioned at the start of that data element; otherwise an error is returned. Searching on wildcards begins from the beginning of the DD list. Wildcards can be used for the tag or reference number (DFTAG\_WILDCARD and DFREF\_WILDCARD) and they match any values.

Hnextread

intn Hnextread(int32 access\_id, uint16 tag, uint16 ref, int origin)

*access\_id* IN: ID of a READ access element

*tag* IN: Tag to search for

*ref* IN: Reference number to search for

*origin* IN: Position at which to start searching

Purpose Locates and positions a read access ID on next occurrence of tag/ref.

Return value Returns SUCCEED (0) if successful and FAIL (-1) otherwise.

Description Searches for the next DD that fits the tag/ref. Wildcards apply. If *origin* is DF\_START, searches from start of DD list; if *origin* is DF\_CURRENT, searches from current position. Searching from the end of the file via DF\_END is not yet implemented.

If the search is successful, then the access element is positioned at the start of that tag/ref; otherwise, the access ID is not modified.

Hstartwrite

int32 Hstartwrite(int32 file\_id, uint16 tag, uint16 ref, int32 length)

*file\_id* IN: ID of file to write to

*tag* IN: Tag to write to

*ref* IN: Reference number to write to

*length* IN: Length of the data element

Purpose Creates or replaces data element with matching tag/ref.

Return value Returns access element ID if successful and FAIL (-1) otherwise.

DescriptionSets up an access element to write a data element. The DD list of the file is searched first; if the tag/ref is found, the data element can be modified. If an object with the corresponding tag/ref is not found, a new one is created.

Hstartaccess

int32 Hstartaccess(int32 file\_id, int16 tag, int16 ref, int32 flags)

*file\_id* IN: ID of file to read/write to

*tag* IN: Tag to read/write to

*ref* IN: Reference number to read/write to

*flags* IN: Access flags for the data element

Purpose Sets up an access element for either reading or writing.

Return value Returns an access element identifier if successful and FAIL (-1) otherwise.

Description Starts up an access element for either read or write access. The data descriptor list for the file is searched first. If the tag/ref is found, it is not replaced; the seek position is presumed to be at zero (0). If the tag/ref is not found, it is created.

Only a finite number of access elements can be active at a given time, so it is important to call Hendaccess whenever you are done using an element.

Hendaccess

int32 Hendaccess(int access\_id)

*access\_id* IN: ID of access element to dispose of

Purpose Disposes of access element for tag/ref.

Return value Returns SUCCEED (0) if successful and FAIL (-1) otherwise.

Description Disposes of an access element. Only a finite number of access elements can be active at a given time, so it is important to call Hendaccess whenever you are done using an element.

When developing new interfaces, a common mistake is to fail to call Hendaccess for all of the elements accessed. When this happens, Hclose will return FAIL and the dump of the error stack (see HEprint below) will tell how many access elements are still active.

This can be difficult problem to debug, as the low levels of the HDF library have no idea who or what opened an access element and forgot to release it. A tedious but effective means of debugging this problem is to annotate with comments the locations where the attached count of a file record is changed. This occurs in the files hfile.c, hblocks.c, and hextelt.c.

Hinquire

intn Hinquire(int32 access\_id, int32 \*pfile\_id, uint16 \*ptag, uint16 \*pref,  
int32 \*plength, int32 \*poffset, int32 \*pposn, int \*paccess,  
int16 \*pspecial)

*access\_id* IN: Access element ID

*pfile\_id* OUT: File ID

*ptag* OUT: Tag of the element pointed to

*pref* OUT: Reference number of the element pointed to

*plength* OUT: Length of the element pointed to

*poffset* OUT: Offset of element in the file

*pposn* OUT: Position pointed to within the data element

*paccess* OUT: Access type of this access element

*pspecial* OUT: Special code

Purpose Returns access information for a data element.

Return value Returns SUCCEED (0) if the access element points to some data element and FAIL (-1) otherwise.

Description Inquires for the statistics of the data element pointed to by the access element. If a piece of information is not needed, a NULL can be sent in for that value. Convenience macros for calls to Hinquire (HQuerypositon, HQuerylength, etc.) are defined in hdf.h.

Hishdf

int32 Hishdf(char \*path)

*path* IN: Complete path and name of file

Purpose Determines whether a file is an HDF file.

Return value Returns TRUE (non-zero) if file is an HDF file and FALSE (0) otherwise.

Description The decision as to whether a file is an HDF file is based solely on the magic number stored in the first four bytes of an HDF file. Hishdf may sometimes identify a file as an HDF file that Hopen is unable to open (e.g., an HDF file with a corrupted DD list).

|  |
| --- |
| **Note:** Hishdf only determines whether a file is an HDF file. It does not verify that the file is readable. |

Hnumber

int Hnumber(int32 file\_id, uint16 tag)

*file\_id* IN: File ID

*tag* IN: Tag to be counted

Purpose Counts the number of occurrences of a tag in a file.

Return value The number of occurrences of a tag in a file.

Hgetlibversion

Hgetlibversion(uint32 \*majorv, uint32 \*minorv, uint32 \*release, char string[])

*majorv* OUT: Major version number

*minorv* OUT: Minor version number

*release* OUT: Release number

*string* OUT: Informational text string

Purpose Gets version information for current HDF library.

Return value Returns SUCCEED (0).

Description Returns the version of the HDF library. The version information is compiled into the HDF library, so it is not necessary to have any open files for this function to execute.

Hgetfileversion

Hgetfileversion(uint32 file\_id, uint32 \*majorv, uint32 \*minorv,   
uint32 \*release, char \*string)

*file\_id* IN: File ID

*majorv* OUT: Major version number

*minorv* OUT: Minor version number

*release* OUT: Release number

*string* OUT: Informational text string

Purpose Gets version information for an HDF file.

Return value Returns SUCCEED (0) if successful and FAIL (-1) otherwise.

Description Returns the HDF version information stored in the given file.

## Reading and Writing Entire Data Elements

Hputelement

int Hputelement(int32 file\_id, uint16 tag, uint16 ref, uint8 \*data,   
int32 length)

*file\_id* IN: File ID

*tag* IN: Tag of data element to put

*ref* IN: Reference number of data element to put

*data* IN: Pointer to buffer

*length* IN: Length of data

Purpose Adds or replaces an element in a file.

Return value Returns SUCCEED (0) if successful and FAIL (-1) otherwise.

Description Writes a new data element or replaces an existing data element in a HDF file. Uses Hwrite and its associated routines.

Hgetelement

int Hgetelement(int32 file\_id, uint16 tag, uint16 ref, uint8 \*data)

*file\_id* IN: ID of the file to read from

*tag* IN: Tag of data element to read

*ref* IN: Reference number of data element to read

*data* OUT: Buffer to read into

Purpose Obtains the data referred to by the passed tag/ref.

Return value Returns SUCCEED (0) if successful and FAIL (-1) otherwise.

DescriptionReads a data element from an HDF file and puts it into the buffer pointed to by *data*. The space allocated for the buffer is assumed to be large enough.

|  |
| --- |
| **Note:** Hgetelement assumes that the buffer is large enough to hold the data being read. It is the user’s responsibility to prevent data loss by ensuring that this is the case. |

## Reading and Writing Part of a Data Element

Hread

int32 Hread(int32 access\_id, int32 length, uint8 \*data)

*access\_id* IN: Read access element ID

*length* IN: Length of segment to read in

*data* OUT: Pointer to data array to read to

Purpose Reads a portion of a data element.

Return value Returns length of segment actually read if successful and FAIL (-1) otherwise.

Description Reads in the next segment in the data element pointed to by the access element. Hread starts at the last position left by an Hread or Hseek call and reads any data that remains in the element up to *length* bytes. If the data element is too short (less than *length* bytes long), Hread reads to the end of the data element.

Hwrite

int32 Hwrite(int32 access\_id, int32 length, uint8 \*data)

*access\_id* IN: Write access element ID

*length* IN: Length of segment to write

*data* IN: Pointer to data to write

Purpose Writes next data segment to data element.

Return value Returns length of segment successfully written and FAIL (-1) otherwise.

Description Writes the data to the data element where the last Hwrite or Hseek stopped.

Hwrite starts at the last position left by an Hwrite or Hseek call, writes up to a specified number of bytes, and leaves the write pointer at the end of the data written. If the space reserved is less than the length to write, then only as much as can fit is written.

It is the user’s responsibility to ensure that no two access elements are writing to the same data element. Note that a user can interlace writes to multiple data elements in the same file.

Hseek

intn Hseek(int32 access\_id, int32 offset, int origin)

*access\_id* IN: Access element ID

*offset* IN: Offset to seek to

*origin* IN: Position to seek from:

DF\_START (0) *offset* from beginning of data element  
DF\_CURRENT (1) *offset* from current position  
DF\_END (2) *offset* from end of data element

PurposeSets the access pointer to an offset within a data element. The next time Hread or Hwrite is called, the read or write occurs from the new position.

Return value Returns SUCCEED (0) if successful and FAIL (-1) otherwise.

Description Sets the position of an access element in a data element so that the next Hread or Hwrite will start from that position. *origin* determines the position from which *offset* should be counted.

This routine fails if the access element is not associated with a data element or if the position sought is outside of the data element.

Seeking from the end of a data element is not currently supported.

## Manipulating Data Descriptors

Hdupdd

int Hdupdd(int32 file\_id, uint16 tag, uint16 ref, uint16 old\_tag,   
uint16 old\_ref)

*file\_id* IN: File ID

*tag* IN: Tag of new data descriptor

*ref* IN: Reference number of new data descriptor

*old\_tag* IN: Tag of data descriptor to duplicate

*old\_ref* IN: Reference number of data descriptor to duplicate

Purpose Generates new references to data that is already referenced from somewhere else.

Return value Returns SUCCEED (0) if successful and FAIL (-1) otherwise.

Description Duplicates a data descriptor so that the new tag/ref points to the same data element pointed to by the old tag/ref.

Hdeldd

int Hdeldd(int32 file\_id, uint16 tag, uint16 ref)

*file\_id* IN: File ID

*tag* IN: Tag of data descriptor to delete

*ref* IN: Reference number of data descriptor to delete

Purpose Deletes a tag/ref from the list of DDs.

Return value Returns SUCCEED (0) if successful and FAIL (-1) otherwise.

DescriptionDeletes the data descriptor of tag/ref from the DD list of the file. This routine is unsafe and may leave a file in a condition that is not usable by some routines. Use with care.

Hnewref

uint16 Hnewref(int32 file\_id)

*file\_id* IN: File ID

Purpose Returns the next available reference number.

Return value Returns the reference number if successful and 0 otherwise.

Description Returns a reference number that can be used with any tag to produce a unique tag/ref. Successive calls to Hnewref will generate a strictly increasing sequence until the highest possible reference number has been returned; then Hnewref will return unused reference numbers starting from 1.

## Managing Special Data Elements

HLcreate

int32 HLcreate(int32 file\_id, uint16 tag, uint16 ref, int32 block\_length,   
int32 number\_blocks)

*file\_id* IN: File ID

*tag* IN: Tag of new data element (or object)

*ref* IN: Reference number of new data element (or object)

*block\_length*IN: Length of blocks to be used

*number\_blocks*IN: Number of blocks to use per linked block record

Purpose*:* Creates a new linked block special data element.

Return value Returns access ID for special data element if successful and FAIL (-1) otherwise.

Description Appending to existing HDF elements was a problem prior to HDF Version 3.2 because HDF objects had to be stored contiguously. When appending, the HDF library forced the user to delete the existing element and rewrite it at the end of the file. HDF Version 3.2 introduced the concept of linked blocks, which allow unlimited appending to existing elements without copying over existing data.

This routine can be used to create an object with the given tag/ref as a linked block element or to promote an existing element to be stored in linked blocks.

Initially, a table is set up to accommodate *number\_blocks* linked blocks for the specified data object. Each block has *block\_length* bytes. If an existing object is being promoted, *block\_length* does not have to be the same size as the original element.

HLcreate returns an active access ID with write permission to the linked block element.

HLsetblockinfo

intn HLsetblockinfo(int32 access\_id, uint32 block\_size, uint32 num\_blocks)

*access\_id* IN: Access record identifier

*block\_size* IN: Block size in bytes

*num\_blocks* IN: Number of linked blocks

Purpose Sets block size and number of blocks for a linked block element.

Return value Returns SUCCEED (0) if successful and FAIL (-1) otherwise.

Description Sets the block size and the number of linked blocks for a linked block data element. Unless reset by this function, block\_size and num\_blocks will have the default values defined in HDF\_APPENDABLE\_BLOCK\_LEN and HDF\_APPENDABLE\_BLOCK\_NUM, respectively.

Passing in the value -1 for either parameter indicates that the respective field is not to be changed.

An error will occur if the value of either parameter is set to 0 or any negative value other than -1.

This routine is used by VSsetblocksize and VSsetnumblocks.

HLgetblockinfo

intn HLgetblockinfo(int32 access\_id, uint32 \*block\_size, uint32 \*num\_blocks)

*access\_id* IN: Access record identifier

*block\_size* OUT: Block size in bytes

*num\_blocks* OUT: Number of linked blocks

Purpose Retrieves block size and number of blocks for a linked block element.

Return value Returns SUCCEED (0) if successful and FAIL (-1) otherwise.

Description Retrieves the block size and the number of linked blocks for a linked block data element.

If no response is desired for either value, block\_size or num\_blocks may be set to NULL.

This routine is used by VSgetblockinfo.

HXcreate

int32 HXcreate(int32 file\_id, uint16 tag, uint16 ref, char \*extern\_file\_name)

*file\_id* IN: file record ID

*tag* IN: Tag of the special data element to create or promote

*ref* IN: Reference number of the special data element to create/promote

*extern\_file\_name*IN: name of the external file to use for the data element

Purpose Creates a new external file special data element.

Return value Returns access ID for special data element if successful and FAIL (-1) otherwise.

Description Creates a new element in an external file or promotes an existing element to be stored in an external file. If an existing element is to be promoted, it is deleted (using Hdeldd) from the original file and copied into the new external file.

Distributing a single object over multiple external files is not currently supported. In addition, one cannot place multiple objects in the same external file.

This routine returns an active access ID with write permission to the external element.

## Data Set Chunking

HMCcreate

int32 HMCcreate(int32 file\_id, uint16 tag, uint16 ref,   
uint8 nlevels, int32 fill\_val\_len, void \*fill\_val,   
HCHUNK\_DEF \*chk\_array)

Purpose

Creates a chunked element.

Description

HMCcreate promotes an HDF element to a chunked element.

The HDF element specified by HMCcreate becomes a chunked element allowing data to be easily appended to the element. Chunk records are stored in a Vdata.

All of the pieces of the chunked element are the same size from the stand point of the element. If compression is used then each chunk is compressed and the compression layer takes care of it as the chunk layer sees each chunks as a seperate HDF object (DFTAG\_CHUNK). The proper compression special header needs to be passed to the compression layer.

The Vdata (chunk table) is made appendable with a linked-block table size of 128.

This routine also creates the chunk cache for the chunked element. The cache is initialized with the physical size of each chunk, the number of chunks in the object, i.e. the object size divided by the chunk size, and the maximum number of chunks to cache in memory. Chunks in the cache are dealt with by their number, i.e. by translating the origin of the chunk to a unique number. The default maximum number of chunks in the cache is set to the number of chunks along the last dimension.

NOTE: The cache itself could be used to cache any object into a number of fixed size chunks so long as the read/write(page-in/page-out) routines know how to deal with getting the correct chunk based on a number.These routines can be found in mcache.c.

Parameters

file\_id IN: File to put chunked element in

tag IN: Tag of element

ref IN: Reference numberof element

nlevels IN: Number of levels of chunks

fill\_val\_len IN: Fill value length in bytes

fill\_val IN: Fill value

chk\_array IN: Structure describing chunk distribution

Return Values

If the chunked element already exists, HMCcreate returns FAIL. Otherwise a new element is created and HMCcreate returns the AID of the newly-created chunked element.

HMCwriteChunk

int32 HMCwriteChunk(int32 access\_id, int32 \*origin, const void \*datap)

Purpose

Writes out exactly one chunk.

Description

HMCwriteChunk writes out exactly one chunk of data to a chunked element.

This function is used to complete whole chunks to the file based on the chunk origin, the position of the chunk in the overall chunk array.

Parameters

access\_id IN: Access AID of the specified chunk.

origin IN: Origin of the chunk to be written.

datap IN: Buffer for the data to be written.

Return Values

Returns the number of bytes written if successful; otherwise returns FAIL.

HMCreadChunk

int32 HMCreadChunk(int32 access\_id, int32 \*origin, void \*datap)

Purpose

Reads exactly one chunk.

Description

HMCreadChunk reads exactly one chunk from a chunked element.

This function is used to read complete chunks from the file based on the chunk origin, the postion of the chunk in the overall chunk array.

Parameters

access\_id IN: Access AID for the specified chunk.

origin IN: Origin of chunk to be read.

datap IN: Buffer for the data to be read.

Return Values

Returns the number of bytes read if successful; otherwise FAIL.

HMCsetMaxcache

int32 HMCsetMaxcache(int32 access\_id, int32 maxcache, int32 flags)

Purpose

Sets themaximum number of chunks to cache.

Description

HMCsetMaxcache sets the maximum number of chunks to cache.

The values set here affects the current object's caching behaviour.

If the chunk cache is full and maxcache is greater then the current maxcache value, then the chunk cache is reset to the new maxcache value, else the chunk cache remains at the current maxcache value.

If the chunk cache is not full, then the chunk cache is set to the new maxcache value only if the new maxcache value is greater than the current number of chunks in the cache.

Use flags arguement of HMC\_PAGEALL if the whole object is to be cached in memory; otherwise pass in zero.

NOTES: This function calls the routine mcache\_set\_maxcache(). The value of maxcache must be greater than 1.

Parameters

access\_id IN: Access AID for the specified chunked element.

maxcache IN: Maximum number of chunks to cache.

flags IN: Valid flags are 0 (zero) and HMC\_PAGEALL.

Returns

Returns the new value of maxcache if successful; otherwise returns FAIL.

HMCPstwrite

int32 HMCPstwrite(accrec\_t \*access\_rec)

Purpose

Opens an access record of a chunked elemnent for writing.

Description

HMCPstwrite calls HMCIstaccess() to fill in the access record for writing.

Parameter

access\_rec IN: Access record to fill in.

Return Values

Returns the AID of the access record if successful; otherwise returns FAIL.

HMCPseek

int32 HMCPseek(accrec\_t \*access\_rec, int32 offset, int origin)

Purpose

Sets the seek position in the chunked element.

Description

HMCPseek sets the seek position in the specified chunked element.

Parameters

access\_rec IN: Access record for the specified chunk.

offset IN: Seek offset.

origin IN: Location from which the offset should be calculated.

Return Values

Returns a positive value if successful; otherwise returns FAIL.

HMCPchunkread

int32 HMCPchunkread(void \*cookie, int32 chunk\_num, void \*datap)

Purpose

Reads a chunk.

Description

Given the chunk number, HMCPchunkread reads in a complete chunk from a chunked element.

This is used as the page-in-chunk routine for the cache.

Only the cache should call this routine.

Parameters

cookie IN: Access record for the desired chunk.

chunk\_num IN: Chunk to be read.

datap OUT: Buffer for data to be read.

Return Values

Returns the number of bytes read if successful; otherwise returns FAIL.

HMCPread

int32 HMCPread(accrec\_t \*access\_rec, int32 length, void \*datap)

Purpose

Reads data from a chunked element.

Description

HMCPread reads in data from a chunked element.

Data is obtained from the cache, which takes care of reading in the proper chunks to satisfy the request.

Parameters

access\_rec IN: Access record for the desired chunk.

length IN: Number of bytes to read.

datap OUT: Buffer for data to be read.

Return Values

Returns the number of bytes read if successful; otherwise returns FAIL.

HMCPchunkwrite

int32 HMCPchunkwrite(void \*cookie, int32 chunk\_num, const void \*datap)

Purpose

Writes out exactly one chunk.

Description

Given the chunk number, HMCPchunkwrite writes a complete chunk to a chunked element.

This is used as the page-out-chunk routine for the cache.

Only the cache should call this routine.

Parameters

cookie IN: Access record for the chunk to be written.

chunk\_num IN: Chunk number.

datap IN: Buffer for the data to be written.

Return Values

Returns the number of bytes written if successful; otherwise returns FAIL.

HMCPwrite

int32 HMCPwrite(accrec\_t \*access\_rec, int32 length, const void \*datap)

Purpose

Writes data to a chunked element.

Description

HMCPwrite writes data to a chunked element.

Data is obtained from the cache, which takes care of obtaining the proper chunks to write to satisfy the request.

The chunks are marked as dirty before being returned to the cache.

Parameters

access\_rec IN: Access record for the chunked element.

length IN: Number of bytes to be written.

datap IN: Buffer for the data to be written.

Return Values

Returns the number of bytes written if successful; otherwise returns FAIL.

HMCPcloseAID

int32 HMCPcloseAID(accrec\_t \*access\_rec)

Purpose

Closes file but keeps AID active.

Description

HMCPcloseAID closes the file currently pointed to by this AID but does not free the AID.

This will flush the chunk cache and free up the special information struct.

This function is called by Hnextread(), which reuses an AID to point to the next object, as requested. If the current object was a chunked object, the chunked information needs to be closed before all reference to it is lost.

NOTE: Direct use of Hnextread() is not recommened since it relies on previous state information.

Parameter

access\_rec IN: Access record of file to close.

Return Values

Returns a positive value if successful; otherwise returns FAIL.

HMCPendaccess

intn HMCPendaccess(accrec\_t \*access\_rec)

Purpose

Closes a chunk element AID.

Description

HMCPendaccess closes the specied AID, freeing up all of the space used to store information about a chunked element and updating the proper records, access\_rec, file\_rec, etc. All relevant information is flushed.

Parameter

access\_rec IN: Access record to close.

Return Values

Returns a positive value if successful; otherwise returns FAIL.

HMCPinfo

int32 HMCPinfo(accrec\_t \*access\_rec, sp\_info\_block\_t \*info\_chunk)

Purpose

Returns information about a chunked element.

Description

HMCPinfo returns information about the given chunked element.

info\_chunk is assumed to be non-NULL.

Parameters

access\_rec IN: access record of access elemement

info\_chunk OUT: Information about the special element.

Return Values

Returns a positive value if successful; otherwise returns FAIL.

HMCPinquire

int32 HMCPinquire(accrec\_t \*access\_rec, int32 \*pfile\_id, uint16 \*ptag,  
uint16 \*pref, int32 \*plength, int32 \*poffset,   
int32 \*pposn, int16 \*paccess, int16 \*pspecial)

Purpose

Inquires for chunked elements.

Description

HMCPinquire returns interesting information about a chunked element.

NULL can be passed for any OUT parameter if the value is not needed.

Parameters

access\_rec IN: Access record of the chunked element for which information is sought.

pfile\_id OUT: File identifier.

ptag OUT: Tag of information record.

pref OUT: Reference number of information record.

plength OUT: Length of element.

poffset OUT: Offset of element -- meaningless.

pposn OUT: Current position in element.

paccess OUT: Access mode.

pspecial OUT: Special code.

Return Values

Returns a positive value if successful; otherwise returns FAIL.

## Development Routines

HDgettagname

char \*HDgettagname(uint16 tag)

*tag* IN: Tag to look up

Purpose Gets a meaningful description of a tag.

Return value Returns a pointer to a string describing this tag or NULL if the tag is unknown.

Description To reduce the amount of duplicated code, this routine can be used to map a tag to a character string containing the name of the tag.

The string returned by this routine is guaranteed to be 30 characters or less.

HDgetspace

void \*HDgetspace(uint32 qty)

*qty* IN: Number of bytes to allocate

Purpose Allocates space.

Return value If successful, returns a pointer to space that was allocated; otherwise returns NULL .

Description Uses an appropriate allocation routine on the local machine to get space.

HDfreespace

void \*HDfreespace(void \*ptr)

*ptr* IN: Pointer to previously-allocated space that is to be freed

Purpose Frees space.

Return value Returns NULL.

Description Uses an appropriate routine on the local machine to free space. This routine is platform dependent.

HDstrncpy

char \*HDstrncpy(register char \*dest, register char \*source, int32 length)

*dest* OUT: Pointer to area to copy string to

*source* IN: Pointer to area to copy string from

*length* IN: Maximum number of bytes to copy

Purpose Copies a string with maximum length *length*.

Return value Returns address of *dest*.

Description Creates a string in *dest* that is at most *length* characters long. The number of characters must *include* the NULL terminator for historical reasons. Hence, if you are working with the string Foo, you must call this copy function with the value 4 (three characters plus the NULL terminator) in *length*.

## Error Reporting

HEprint

void HEprint(FILE \*stream, int32 level)

*stream* IN: Stream to print error messages on

*level* IN: Level of the error stack to print

Purpose Prints information on the error stack.

Return value Has no return value.

Description Prints information on reported errors. If *level* is zero, all of the errors currently on the error stack are printed. Output from this function is sent to the file pointed to by *stream*.

The following information is printed:

* An ASCII description of the error
* The reporting routine
* The reporting routine’s source file name
* The line at which the error was reported

If the programmer has supplied extra information by means of HEreport, this information is printed as well.

HEclear

void HEclear(void)

Purpose Clears all information on reported errors off of the error stack.

Return value Has no return value.

Description Clears all of the information off of the error stack.

HERROR

void HERROR(int16 number)

*number* IN: Error number

Purpose Reports an error.

Return value Has no return value.

DescriptionReports an error. Any function calling HERROR must have a variable FUNC which points to a string containing the name of the function.

HERROR is implemented as a macro.

HEreport

void HEreport(char \*format, ....)

*format* IN: printf-style format and arguments

Purpose Provides extra information to the error reporting routines.

Return value Has no return value.

Description Provides further annotation to an error report. Only one such annotation is remembered for each error report. The arguments to this routine follow the style of printf.

Consider the following example from hfile.c:

char \*FUNC = "Hclose";

....

if (file\_rec->attach > 0) {

file\_rec->refcount++;

HERROR(DFE\_OPENAID);

HEreport("There are still %d active aids attached", file\_rec->attach);

return FAIL;

}

## Other

Hsync

int Hsync(int32 file\_id)

*file\_id* IN: ID of the file to synchronize

Purpose Synchronizes on-disk HDF file with image in memory.

Return value Returns SUCCEED.

Description Hsync is not included in the current HDF library release because the on-disk representation of an HDF file is always the same as its in-memory representation. Hsync will be provided when future releases implement buffering schemes.