Copyright Notice and Statement for NCSA Hierarchical Data Format (HDF) Software Library and Utilities

Copyright 1988-2003 The Board of Trustees of the University of Illinois

All rights reserved.

Contributors: National Center for Supercomputing Applications (NCSA) at the University of Illinois, Fortner Research, Unidata Program Center (netCDF), The Independent JPEG Group (JPEG), Jean-loup Gailly and Mark Adler (gzip), and Digital Equipment Corporation (DEC).

Redistribution and use in source and binary forms, with or without modification, are permitted for any purpose (including commercial purposes) provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the following disclaimer in the documentation and/or materials provided with the distribution.
3. In addition, redistributions of modified forms of the source or binary code must carry prominent notices stating that the original code was changed and the date of the change.
4. All publications or advertising materials mentioning features or use of this software must acknowledge that it was developed by the National Center for Supercomputing Applications at the University of Illinois, and credit the Contributors.
5. Neither the name of the University nor the names of the Contributors may be used to endorse or promote products derived from this software without specific prior written permission from the University or the Contributors.

Disclaimer

THIS SOFTWARE IS PROVIDED BY THE UNIVERSITY AND THE CONTRIBUTORS "AS IS" WITH NO WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED. In no event shall the University or the Contributors be liable for any damages suffered by the users arising out of the use of this software, even if advised of the possibility of such damage.

Trademarks

Sun is a registered trademark, and Sun Workstation, Sun/OS and Solaris are trademarks of Sun Microsystems Inc. / UNIX is a registered trademark of X/Open / VAX and VMS are trademarks of Digital Equipment Corporation / Macintosh is a trademark of Apple Computer, Inc. / CRAY and UNICOS are registered trademarks of Silicon Graphics, Inc. / IBM PC is a registered trademark of International Business Machines Corporation / MS-DOS is a registered trademark of Microsoft Corporation.

The SZIP Science Data Lossless Compression Program is Copyright (C) 2001 Science & Technology Corporation @ UNM. All rights released. Copyright (C) 2003 Lowell H. Miles and Jack A. Venbrux. Licensed to ICS Corp. for distribution by the University of Illinois' National Center for Supercomputing Applications as a part of the HDF data storage and retrieval file format and software library products package. All rights reserved. Do not modify or use for other purposes. See for further information regarding terms of use.

NCSA Contacts

Mail user feedback, bug reports, and questions to:

NCSA Scientific Data Technologies
HDF Group
152 Computing Applications Bldg.
605 E. Springfield Ave.
Champaign, IL 61820-5518

Send electronic correspondence and bug reports to the following:

hdfhelp@ncsa.uiuc.edu

Internet access

HDF is available without charge from NCSA's anonymous FTP server:
ftp.ncsa.uiuc.edu/HDF/HDF/

It is also accessible through the HDF Group's World Wide Web home page:
http://hdf.ncsa.uiuc.edu/
# Table of Contents

1.1 Overview of the HDF Interfaces .................................................. 1-1  
1.2 Low-Level Interface ...................................................................... 1-1  
1.3 Multifile Application Interfaces .................................................. 1-2  
1.3.1 Scientific Data Sets: SD Interface .............................................. 1-2  
1.3.2 Annotations: AN Interface ......................................................... 1-2  
1.3.3 General Raster Images: GR Interface ......................................... 1-2  
1.3.4 Scientific Data Sets: netCDF Interface ....................................... 1-2  
1.3.5 Vdata: The VS Interface ............................................................ 1-3  
1.3.6 Vdata Query: VSQ Interface ....................................................... 1-3  
1.3.7 Vdata Fields: VF Interface ........................................................ 1-3  
1.3.8 Vgroups: V Interface ............................................................... 1-3  
1.3.9 Vdata/Vgroups: VH Interface ...................................................... 1-3  
1.3.10 Vgroup Inquiry: VQ Interface ................................................... 1-3  
1.4 Single-File Application Interfaces ................................................. 1-3  
1.4.1 24-bit Raster Image Sets: DF24 Interface ................................... 1-4  
1.4.2 8-bit Raster Image Sets: DFR8 Interface .................................... 1-4  
1.4.3 Palettes: DFP Interface ............................................................. 1-4  
1.4.4 Scientific Data Sets: DFSD Interface ......................................... 1-4  
1.4.5 Annotations: DFAN Interface .................................................... 1-4  
1.5 FORTRAN-77 and C Language Issues ......................................... 1-5  
1.5.1 FORTRAN-77-to-C Translation ............................................... 1-5  
1.5.2 Case Sensitivity ........................................................................ 1-5  
1.5.3 Name Length ............................................................................ 1-5  
1.5.4 Header Files ............................................................................. 1-5  
1.5.5 Data Type Specifications .......................................................... 1-5  
1.5.6 String and Array Specifications ............................................... 1-6  
1.5.7 FORTRAN-77, ANSI C and K&R C ......................................... 1-6  
1.6 Error Codes ................................................................................. 1-7  
2.1 Reference Section Overview ......................................................... 2-11  
ANannlen/afannlen ........................................................................... 2-13  
ANannlist/afannlist .......................................................................... 2-14  
ANatype2tag/afatypetag ................................................................... 2-15  
ANcreate/afcreate ............................................................................ 2-16  
ANcreatel/afcreatel .......................................................................... 2-17  
ANend/afend ...................................................................................... 2-18  
ANendaccess/afendaccess ............................................................... 2-19  
ANfileinfo/affileinfo .......................................................................... 2-20  
ANGet_tagref/afgettagref ............................................................... 2-21  
ANid2tagref/afidtagref .................................................................... 2-22  
ANnumann/afnumann ........................................................................ 2-23  
ANreadann/afreadann ........................................................................ 2-24  
ANSel ect/afselect ............................................................................... 2-25  
ANstart/afstart .................................................................................. 2-26  
ANTag2type/afntagatype ................................................................. 2-27  
ANTagref2id/aftagrefid ................................................................. 2-28  
ANwriteann/afwriteann .................................................................... 2-29  
G Rattrinfo/mgatinf ........................................................................... 2-31
Table of Contents

GRcreate/mgcreat .................................................. 2-32
GRend/mgend ........................................................ 2-34
GRendaccess/mgendac .............................................. 2-35
GRfileinfo/mgffinfo .................................................. 2-36
GRfindattr/mgffndat ............................................... 2-37
GRgetattr/mggnatt/mgscatt ...................................... 2-38
GRgetchunkinfo/mgffichnk ........................................ 2-39
GRgetcompress/mgfgcompress ...................................... 2-41
GRgetinfo/mgfflinf ................................................. 2-42
GRgetlutm/mgfltlid ................................................ 2-43
GRgetlutf/info/mgflinf ........................................... 2-44
GRgetnluts/mgfnluts .............................................. 2-45
GRridtofile/mgfid2rf .............................................. 2-46
GRluftofe/mglid2rf ................................................. 2-47
GRnametoindex/mgn2ndx .......................................... 2-48
GRreadchunk/mgrchnk/mgrcchunk .................................. 2-49
GRreadimage/mgrdimg/mgrcimg ................................... 2-50
GRreadlutm/mgrflutm/mgrclut .................................. 2-52
GRrefltoindex/mgrfidx ........................................... 2-53
GRreqimagei/mgrrimil ............................................. 2-54
GRrequtil/mgrlutil ................................................ 2-55
GRselect/mgselct .................................................. 2-56
GRsetattr/mgsnatt/mgsct ......................................... 2-57
GRsetcompress/mgsccompress .................................... 2-59
GRsetchunk/mgschknk .............................................. 2-61
GRsetchunkcache/mgscschknk ................................... 2-64
GRsetexternalfile/mgsxfil ....................................... 2-65
GRstart/mgstart .................................................... 2-66
GRwritechunk/mgwchknk/mgwchck ................................ 2-67
GRwriteimage/mgrwimmg/mgwimg .................................. 2-68
GRwritelut/mgwrlutm/mgwclut .................................. 2-70
Hclos/hclos .......................................................... 2-73
Hgetfileversion/hgffilver ....................................... 2-74
Hgetlibversion/hglbbiver ........................................ 2-75
Hshdf/hshdf .......................................................... 2-76
Hopen/hopen .......................................................... 2-77
HCgetcompress ........................................................ 2-78
HDdnot_atexit/hddontatexit ..................................... 2-79
HEprint/heprintf/heprintf ...................................... 2-80
HEstring/hestringf ............................................... 2-81
HXsetcreatedir/hxiscdir ....................................... 2-82
HXsetdir/hxisdir .................................................. 2-83
SDattrinfo/sfgainfo ............................................... 2-85
SDcheckempty/sfcheckemty ....................................... 2-86
SDcreate/sfcreate .................................................. 2-87
SDdiminfo/sfordinfo ............................................... 2-89
SDend/sfend .......................................................... 2-90
SDendaccess/sfsendacc .......................................... 2-91
SDfileinfo/sffinfo .................................................. 2-92
SFindattr/sffattr ................................................... 2-93
SDgetca/sfgcal ....................................................... 2-94
SDgetchunkinfo/sfichnk .......................................... 2-95
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDgetcompress/sfgcompress</td>
<td>2-97</td>
</tr>
<tr>
<td>SDgetdatastrs/sfgdstr</td>
<td>2-98</td>
</tr>
<tr>
<td>SDgetdimid/sfdimid</td>
<td>2-99</td>
</tr>
<tr>
<td>SDgetdimscale/sfgdscale</td>
<td>2-100</td>
</tr>
<tr>
<td>SDgetdimstrs/sfgdmsstr</td>
<td>2-101</td>
</tr>
<tr>
<td>SDgetfillvalue/sfgfill/sfgcfill</td>
<td>2-102</td>
</tr>
<tr>
<td>SDgetinfo/sfginfo</td>
<td>2-103</td>
</tr>
<tr>
<td>SDgetrange/sfgrange</td>
<td>2-104</td>
</tr>
<tr>
<td>SDdidore/sfid2ref</td>
<td>2-105</td>
</tr>
<tr>
<td>SDiscoordvar/sfiscvar</td>
<td>2-106</td>
</tr>
<tr>
<td>SDisdimval_bwcomp/sfsdmvc</td>
<td>2-107</td>
</tr>
<tr>
<td>SDisrecord/sfisrcrd</td>
<td>2-108</td>
</tr>
<tr>
<td>SDnametoindex/sfn2index</td>
<td>2-109</td>
</tr>
<tr>
<td>SDreadattr/sfratt/sfrcatt</td>
<td>2-110</td>
</tr>
<tr>
<td>SDreadchunk/sfrchnk/sfrcchnk</td>
<td>2-111</td>
</tr>
<tr>
<td>SDreaddata/sfrdata/sfrcdatad</td>
<td>2-112</td>
</tr>
<tr>
<td>SDrefindex/sfref2index</td>
<td>2-114</td>
</tr>
<tr>
<td>SDselect/sfselect</td>
<td>2-115</td>
</tr>
<tr>
<td>SDsetattr/sfsnatt/sfscatt</td>
<td>2-116</td>
</tr>
<tr>
<td>SDsetblocksize/sfsbslz</td>
<td>2-118</td>
</tr>
<tr>
<td>SDsetcal/sfscal</td>
<td>2-119</td>
</tr>
<tr>
<td>SDsetchunk/sfschnk</td>
<td>2-120</td>
</tr>
<tr>
<td>SDsetchunkcache/sfsccchnk</td>
<td>2-123</td>
</tr>
<tr>
<td>SDsetcompress/sfscompress</td>
<td>2-124</td>
</tr>
<tr>
<td>SDsetdatastrs/sfsdstr</td>
<td>2-126</td>
</tr>
<tr>
<td>SDsetdimname/sfsdimname</td>
<td>2-127</td>
</tr>
<tr>
<td>SDsetdimscale/sfsdscale</td>
<td>2-128</td>
</tr>
<tr>
<td>SDsetdimstrs/sfsdmstr</td>
<td>2-129</td>
</tr>
<tr>
<td>SDsetdimval_comp/sfsdmvc</td>
<td>2-130</td>
</tr>
<tr>
<td>SDsetexternalfile/sfsextf</td>
<td>2-131</td>
</tr>
<tr>
<td>SDsetfillmode/sfsflmd</td>
<td>2-132</td>
</tr>
<tr>
<td>SDsetfillvalue/sfsfill/sfsfcall</td>
<td>2-133</td>
</tr>
<tr>
<td>SDsetnbitdataset/sfsnbit</td>
<td>2-134</td>
</tr>
<tr>
<td>SDsetrange/sfsrange</td>
<td>2-136</td>
</tr>
<tr>
<td>SDstart/sfstart</td>
<td>2-137</td>
</tr>
<tr>
<td>SDwritechunk/sfwchnk/sfwcchnk</td>
<td>2-138</td>
</tr>
<tr>
<td>SDwritedata/sfwdata/sfwcdatad</td>
<td>2-139</td>
</tr>
<tr>
<td>Vaddtagref/vfadtr</td>
<td>2-141</td>
</tr>
<tr>
<td>Vattach/vfatch</td>
<td>2-142</td>
</tr>
<tr>
<td>Vattrinfo/vfainfo</td>
<td>2-143</td>
</tr>
<tr>
<td>Vdelete/vdelete</td>
<td>2-144</td>
</tr>
<tr>
<td>Vdeleteetagref/vfdatr</td>
<td>2-145</td>
</tr>
<tr>
<td>Vdetach/vfdtch</td>
<td>2-146</td>
</tr>
<tr>
<td>Vend/vfend</td>
<td>2-147</td>
</tr>
<tr>
<td>Viind/vfind</td>
<td>2-148</td>
</tr>
<tr>
<td>Viindattr/vfidatt</td>
<td>2-149</td>
</tr>
<tr>
<td>Viindclass/vfndcls</td>
<td>2-150</td>
</tr>
<tr>
<td>Vflocate/vflloc</td>
<td>2-151</td>
</tr>
<tr>
<td>Vgetattr/vfgnatt/vfgcatt</td>
<td>2-152</td>
</tr>
<tr>
<td>Vgetclass/vfgcls</td>
<td>2-153</td>
</tr>
<tr>
<td>Vgetid/vfgid</td>
<td>2-154</td>
</tr>
<tr>
<td>Vgetname/vfgnam</td>
<td>2-155</td>
</tr>
</tbody>
</table>

12/30/03
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vgetnext/vfgnxt</td>
<td>2-156</td>
</tr>
<tr>
<td>Vgetagref/vfgtr</td>
<td>2-157</td>
</tr>
<tr>
<td>Vgetagrefs/vfgtrrs</td>
<td>2-158</td>
</tr>
<tr>
<td>Vgetversion/vfgver</td>
<td>2-159</td>
</tr>
<tr>
<td>Vinqagref/vfinqtr</td>
<td>2-160</td>
</tr>
<tr>
<td>Vinsert/vfinsrt</td>
<td>2-161</td>
</tr>
<tr>
<td>Vset/vfs</td>
<td>2-162</td>
</tr>
<tr>
<td>Vset/vfs</td>
<td>2-163</td>
</tr>
<tr>
<td>Vget/vfsvs</td>
<td>2-164</td>
</tr>
<tr>
<td>Vlone/vfalone</td>
<td>2-165</td>
</tr>
<tr>
<td>Vnattrs/vfnatts</td>
<td>2-166</td>
</tr>
<tr>
<td>Vnrefs/vfnrefs</td>
<td>2-167</td>
</tr>
<tr>
<td>Vngang/vfngtr</td>
<td>2-168</td>
</tr>
<tr>
<td>Vsetattr/vfnatt/vfgang</td>
<td>2-169</td>
</tr>
<tr>
<td>Vsetclass/vfscls</td>
<td>2-170</td>
</tr>
<tr>
<td>Vsetname/vfsnam</td>
<td>2-171</td>
</tr>
<tr>
<td>Vstart/vfstart</td>
<td>2-172</td>
</tr>
<tr>
<td>VHmakegroup/vhfgroup</td>
<td>2-173</td>
</tr>
<tr>
<td>VQuery/vfqref</td>
<td>2-174</td>
</tr>
<tr>
<td>VQuerytag/vqtag</td>
<td>2-175</td>
</tr>
<tr>
<td>VFieldsize/vffesiz</td>
<td>2A-177</td>
</tr>
<tr>
<td>VFieldsize/vffesiz</td>
<td>2A-178</td>
</tr>
<tr>
<td>VFieldname/vfname</td>
<td>2A-179</td>
</tr>
<tr>
<td>VFieldorder/vffordr</td>
<td>2A-180</td>
</tr>
<tr>
<td>VFieldtype/vfctype</td>
<td>2A-181</td>
</tr>
<tr>
<td>VFnfields/vfnfields</td>
<td>2A-182</td>
</tr>
<tr>
<td>VSQuerycount/vsqfnelt</td>
<td>2A-183</td>
</tr>
<tr>
<td>VSQueryfields/vsqfields</td>
<td>2A-184</td>
</tr>
<tr>
<td>VSQueryinterlace/vsqinter</td>
<td>2A-185</td>
</tr>
<tr>
<td>VSQueryname/vsqname</td>
<td>2A-186</td>
</tr>
<tr>
<td>VSQueryref/vsqref</td>
<td>2A-187</td>
</tr>
<tr>
<td>VSQuerytag/vsqtag</td>
<td>2A-188</td>
</tr>
<tr>
<td>VSQueryvsize/vsqfsize</td>
<td>2A-189</td>
</tr>
<tr>
<td>VHstoredata/vhfs/vhfsd</td>
<td>2-190</td>
</tr>
<tr>
<td>VHstoredatam/vhfsdm/vhfsd</td>
<td>2-191</td>
</tr>
<tr>
<td>VSAppendable/vsapp</td>
<td>2-192</td>
</tr>
<tr>
<td>VSAttach/vsfatch</td>
<td>2-193</td>
</tr>
<tr>
<td>VSAttrinfo/vsfainfo</td>
<td>2-194</td>
</tr>
<tr>
<td>VSDel/vsfdelete</td>
<td>2-195</td>
</tr>
<tr>
<td>VSDetach/vsfdetach</td>
<td>2-196</td>
</tr>
<tr>
<td>VSElts/vsfelts</td>
<td>2-197</td>
</tr>
<tr>
<td>VSDefine/vsfndef</td>
<td>2-198</td>
</tr>
<tr>
<td>VSExist/vsfexist</td>
<td>2-199</td>
</tr>
<tr>
<td>VSFind/vsffind</td>
<td>2-200</td>
</tr>
<tr>
<td>VSFindattr/vsffindat</td>
<td>2-201</td>
</tr>
<tr>
<td>VSFindclass/vsfclass</td>
<td>2-202</td>
</tr>
<tr>
<td>VSIndex/vsfindex</td>
<td>2-203</td>
</tr>
<tr>
<td>VSattr/vsfnattr</td>
<td>2-204</td>
</tr>
<tr>
<td>VSpack/vsfpak/vsfnpak</td>
<td>2-205</td>
</tr>
<tr>
<td>VSGetattr/vsfgetat/vsfgetat</td>
<td>2-206</td>
</tr>
<tr>
<td>VSGetblockinfo/vsfblockinfo</td>
<td>2-207</td>
</tr>
<tr>
<td>VSGetclass/vsfclass</td>
<td>2-208</td>
</tr>
<tr>
<td>VSGetclass/vsfclass</td>
<td>2-209</td>
</tr>
<tr>
<td>Function Call</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>DFPnpals/dppnpals</td>
<td>2-272</td>
</tr>
<tr>
<td>DFPtruptal/dpppal</td>
<td>2-273</td>
</tr>
<tr>
<td>DFPreadref/dprref</td>
<td>2-274</td>
</tr>
<tr>
<td>DFPrestart/dpres</td>
<td>2-275</td>
</tr>
<tr>
<td>DFPtrwref/dpwref</td>
<td>2-276</td>
</tr>
<tr>
<td>DFKNTsize</td>
<td>2-277</td>
</tr>
<tr>
<td>DFUpfimoimage/dupf2im</td>
<td>2-279</td>
</tr>
<tr>
<td>DFNAddfds/daafds</td>
<td>2-281</td>
</tr>
<tr>
<td>DFNAddfidd/daafid</td>
<td>2-282</td>
</tr>
<tr>
<td>DFNclear/daclear</td>
<td>2-283</td>
</tr>
<tr>
<td>DFNgetdesc/dagdesc</td>
<td>2-284</td>
</tr>
<tr>
<td>DFNgetdesclen/dagdlen</td>
<td>2-285</td>
</tr>
<tr>
<td>DFNgetfds/dagfds</td>
<td>2-286</td>
</tr>
<tr>
<td>DFNgetfdslen/dagfidsl</td>
<td>2-287</td>
</tr>
<tr>
<td>DFNgetfidslen/dagfidsl</td>
<td>2-288</td>
</tr>
<tr>
<td>DFNgetfilen/dagfifl</td>
<td>2-289</td>
</tr>
<tr>
<td>DFNgetlabel/daglabel</td>
<td>2-290</td>
</tr>
<tr>
<td>DFNgetlabelen/daglabelen</td>
<td>2-291</td>
</tr>
<tr>
<td>DFNlablist/dallist</td>
<td>2-292</td>
</tr>
<tr>
<td>DFNlastref/dalref</td>
<td>2-293</td>
</tr>
<tr>
<td>DFNputdesc/dapdesc</td>
<td>2-294</td>
</tr>
<tr>
<td>DFNputlabel/daplabeled</td>
<td>2-295</td>
</tr>
<tr>
<td>DFSDaaddata/dsdadata</td>
<td>2-297</td>
</tr>
<tr>
<td>DFSDclear/dsclear</td>
<td>2-298</td>
</tr>
<tr>
<td>DFSDendslab/dseslab</td>
<td>2-299</td>
</tr>
<tr>
<td>DFSDendslice/dseslsel</td>
<td>2-300</td>
</tr>
<tr>
<td>DFSDgetcal/dsgcal</td>
<td>2-301</td>
</tr>
<tr>
<td>DFSDgetdata/dsgdata</td>
<td>2-302</td>
</tr>
<tr>
<td>DFSDgetdata4l/dsgdata4l</td>
<td>2-303</td>
</tr>
<tr>
<td>DFSDgetdatast/r/dsgdatast</td>
<td>2-304</td>
</tr>
<tr>
<td>DFSDgetdimlen/dsgdiln</td>
<td>2-305</td>
</tr>
<tr>
<td>DFSDgetdms/dsgdms</td>
<td>2-306</td>
</tr>
<tr>
<td>DFSDgetdmscale/dsgdisc</td>
<td>2-307</td>
</tr>
<tr>
<td>DFSDgetdmsstsr/dsgdmsstsr</td>
<td>2-308</td>
</tr>
<tr>
<td>DFSDgetfillvalue/dsgfillvalue</td>
<td>2-309</td>
</tr>
<tr>
<td>DFSDgetNT/dsgnt</td>
<td>2-310</td>
</tr>
<tr>
<td>DFSDgetrange/dsgrange</td>
<td>2-311</td>
</tr>
<tr>
<td>DFSDgetslice/dsgslice</td>
<td>2-312</td>
</tr>
<tr>
<td>DFSDlastref/dsref</td>
<td>2-313</td>
</tr>
<tr>
<td>DFSDndatasets/dsnum</td>
<td>2-314</td>
</tr>
<tr>
<td>DFSDpcre32sdg/dsp32sd</td>
<td>2-315</td>
</tr>
<tr>
<td>DFSDpudpdata/dspdata</td>
<td>2-316</td>
</tr>
<tr>
<td>DFSDputslice/dspslice</td>
<td>2-317</td>
</tr>
<tr>
<td>DFSDreadref/dsref</td>
<td>2-318</td>
</tr>
<tr>
<td>DFSDreadslab/drsrlab</td>
<td>2-319</td>
</tr>
<tr>
<td>DFSDrestart/dsfirst</td>
<td>2-320</td>
</tr>
<tr>
<td>DFSDsetcal/dsscal</td>
<td>2-321</td>
</tr>
<tr>
<td>DFSDsetdataas/dssdataas</td>
<td>2-322</td>
</tr>
<tr>
<td>DFSDsetdms/dssdms</td>
<td>2-323</td>
</tr>
<tr>
<td>DFSDsetdmscale/dssdisc</td>
<td>2-324</td>
</tr>
<tr>
<td>DFSDsetdmsstsr/dssdmsstsr</td>
<td>2-325</td>
</tr>
<tr>
<td>DFSDsetfillvalue/dssfillvalue</td>
<td>2-326</td>
</tr>
<tr>
<td></td>
<td>2-327</td>
</tr>
</tbody>
</table>
3.1 Definition List Overview ................................................................. 3-369
Introduction to the HDF APIs

1.1 Overview of the HDF Interfaces

The HDF library structure consists of two interface layers and one application layer built upon a physical file format. (See Figure 1a.) The first layer, or the low-level interface, is generally reserved for software developers because it provides support for low-level details such as file I/O, error handling, and memory management. The second layer, containing the single and multi-file application interfaces, consists of a set of interfaces designed to simplify the process of storing and accessing data. The single-file interfaces operate on one file at a time, whereas the multi-file interfaces can operate on several files simultaneously. The highest HDF layer includes various NCSA and commercial applications and a collection of command-line utilities that operate on HDF files or the data objects they contain.

1.2 Low-Level Interface

This is the layer of HDF reserved for software developers and provides routines for error handling, file I/O, memory management, and physical storage. These routines are prefaced with 'H'. For a more detailed discussion of the low-level interface, consult the HDF Specifications and Developer’s Guide from the HDF WWW home page at http://hdf.ncsa.uiuc.edu/.

The low-level interface provides a collection of routines that are prefaced with either 'H', 'HE', or 'HX'. The H routines are for managing HDF files. The HE routines provide error handleings. The HX routines are for managing HDF external files.
Prior to HDF version 3.2, all low-level routines began with the prefix 'DF'. As of HDF version 3.3, the DF interface was no longer recommended for use. It is only supported to maintain backward compatibility with programs and files created under earlier versions of the HDF library.

1.3 Multifile Application Interfaces

The HDF multifile interfaces are designed to allow operations on more than one file and more than one data object at the same time. The multifile interfaces provided are AN, GR, SD, VS, VSQ, VF, V, and VH. The AN interface is the multifile version of the DFAN annotation interface. The GR interface is the multifile version of the 8- and 24-bit raster image interfaces. The SD interface is the multifile version of the scientific data set interface. The VS, VSQ, and VF interfaces support the vdata model. The V and VH interfaces provide support for the vgroup data model.

Like the single-file interfaces, the multifile interfaces are built upon the low-level H routines. Unlike single-file operations, operations performed via a multifile interface are not implicitly preceded by Hopen and followed by Hclose. Instead, each series of operations on a file must be preceded by an explicit call to open and close the file. Once the file is opened, it remains open until an explicit call is made to close it. This process allows operations on more than one file at a time.

1.3.1 Scientific Data Sets: SD Interface

The scientific data set interface provides a collection of routines for reading and writing arrays of data. Multidimensional arrays accompanied by a record of their dimension and number type are called scientific data sets. Under the multifile interface, scientific data sets may include predefined or user defined attribute records. Each attribute record is optional and describes a particular facet of the environment from which the scientific data was taken.

The names of the routines in the multifile scientific data set interface are prefaced by 'SD'. The equivalent FORTRAN-77 routine names are prefaced by ’sf’.

1.3.2 Annotations: AN Interface

The purpose of the AN multifile annotation interface is to permit concurrent operations on a set of annotations that exist in more than one file. Annotations consist of labels and descriptions.

The C routine names of the multifile annotation interface are prefaced by the string ’AN’ and the FORTRAN-77 routine names are prefaced by ’af’.

1.3.3 General Raster Images: GR Interface

The routines in the GR interface provide multifile operations on general raster image data sets.

The C routine names in the general raster interface have the prefix ’GR’ and the equivalent FORTRAN-77 routine names are prefaced by ’mg’.

1.3.4 Scientific Data Sets: netCDF Interface

The SD interface is designed to be as compatible as possible with netCDF, an interface developed by the Unidata Program Center. Consequently, the SD interface can read files written by the netCDF interface, and the netCDF interface (as implemented in HDF) can read both netCDF files and HDF files that contain scientific data sets.

Further information regarding the netCDF interface routines and their equivalents in the HDF interface can be found in the HDF User's Guide. Additional information on the netCDF interface can be found in the netCDF User's Guide available by anonymous ftp from unidata.ucar.edu.
1.3.5 Vdata: The VS Interface

The VS interface provides a collection of routines for reading and writing customized tables. Each table is comprised of a series of records whose values are stored in fixed length fields. In addition to its records, a vdata may contain four kinds of identifying information: a name, class, data type and a number of field names.

Routines in the VS interface are prefaced by 'VS'. The equivalent FORTRAN-77 routine names are prefixed by 'vsf'.

1.3.6 Vdata Query: VSQ Interface

The VSQ interface provides a collection of routines for inquiring about existing vdata. These routines provide information such as the number of records in a vdata, its field names, number types, and name. All routines in the VSQ interface are prefaced by 'VSQ'. The equivalent FORTRAN-77 routine names are prefixed by 'vsq'.

1.3.7 Vdata Fields: VF Interface

The VF interface provides a collection of routines for inquiring about the fields in an existing vdata. These routines provide information such as the field name, size, order, and number type.

All routines in the VF interface are prefaced by 'VF'. There are no equivalent FORTRAN-77 functions.

1.3.8 Vgroups: V Interface

The vgroup interface provides a collection of routines for grouping and manipulating HDF data objects in the file. Each vgroup may contain one or more vdatas, vgroups, or other HDF data objects. In addition to its members, a vgroup may also be given a name and a class.

Every routine name in the vgroup interface are prefaced by 'V'. The equivalent FORTRAN-77 routine names are prefixed by 'vf'.

1.3.9 Vdata/Vgroups: VH Interface

The high-level VH interface provides a collection of routines for creating simple vdatas and vgroups with a single function call. All routines in this interface are prefaced by 'VH'. The equivalent FORTRAN-77 routine names are prefixed by 'vh'.

1.3.10 Vgroup Inquiry: VQ Interface

The high-level VQ interface provides one routine that returns tag information from a specified vgroup, and one routine that returns reference number information from a specified vgroup. All C routine names in this interface are prefaced by 'VQ'. The equivalent Fortran-77 routine names are prefixed by 'vq'.

1.4 Single-File Application Interfaces

The HDF single-file application interfaces include several independent modules each is designed to simplify the process of storing and accessing a specific type of data. These interfaces support the 8-bit raster image (DFR8), 24-bit raster image (DF24), palette (DFP), scientific data (DFSD), and annotation (DFAN) models. All single-file interfaces are built upon the H routines - unless otherwise specified, all the low-level details can be ignored.
1.4.1 24-bit Raster Image Sets: DF24 Interface

The HDF 24-bit raster interface provides a collection of routines for managing 24-bit raster image sets. A 24-bit raster image set is comprised of a 24-bit raster image array and its accompanied dimension record. Raster image sets may also include a palette.

The names of the routines in the 24-bit raster interface are prefaced by 'DF24'. The equivalent FORTRAN-77 routine names are prefaced by 'd2'.

1.4.2 8-bit Raster Image Sets: DFR8 Interface

The HDF 8-bit raster interface provides a collection of routines for managing 8-bit raster image sets. An 8-bit raster image set is comprised of an 8-bit raster image array and its accompanied dimension record. Raster image sets may also include a palette.

Every function in the 8-bit raster interface begins with the prefix 'DFR8'. The equivalent FORTRAN-77 functions use the prefix 'd8'.

1.4.3 Palettes: DFP Interface

The HDF palette interface provides a collection of routines for managing palette data. This interface is most often used for working with multiple palettes stored in a single file or palettes not specifically assigned to a raster image.

The names of the routines in the palette interface are prefaced by 'DFP'. The equivalent FORTRAN-77 routine names are prefaced by 'dp'.

1.4.4 Scientific Data Sets: DFSD Interface

There are two HDF interfaces that support multidimensional arrays: the single-file DFSD interface described here, which permits access to only one file at a time, and the newer multifile SD interface, which permits simultaneous access to more than one file. The existence of the single-file scientific data set interface is simply to support backward compatibility for previously created files and applications. It is recommended that the multifile scientific data set interface is to be used where possible.

The single-file scientific data set interface provides a collection of routines for reading and writing arrays of data. A scientific data set is comprised of a scientific data array and its accompanied rank, name and number type. Scientific data sets may also include predefined attribute records.

The names of the routines in the single-file scientific data set interface are prefaced by 'DFSD'. The equivalent FORTRAN-77 routine names are prefaced by 'ds'.

1.4.5 Annotations: DFAN Interface

The single-file annotation interface provides a collection of routines for reading and writing text strings assigned to HDF data objects or files. Annotations consist of labels and descriptions.

The names of the routines in the single-file annotation interface are prefaced by 'DFAN'. The equivalent FORTRAN-77 routine names are prefaced by 'da'.

National Center for Supercomputing Applications
1.5 FORTRAN-77 and C Language Issues

In order to make the FORTRAN-77 and C versions of each routine as similar as possible, some compromises have been made in the process of simplifying the interface for both programming languages.

1.5.1 FORTRAN-77-to-C Translation

Nearly all of the HDF library code is written in C. The Fortran HDF API routines translate all parameter data types to C data types, then call the C routine that performs the main function. For example, \texttt{d8aimg} is the FORTRAN-77 equivalent for \texttt{DFR8addimage}. Calls to either routine execute the same C code that adds an 8-bit raster image to an HDF file - see the following figure.

1.5.2 Case Sensitivity

FORTRAN-77 identifiers generally are not case sensitive, whereas C identifiers are. Although all of the FORTRAN-77 routines shown in this manual are written in lower case, FORTRAN-77 programs can generally call them using either upper- or lower-case letters without loss of meaning.

1.5.3 Name Length

Because some FORTRAN-77 compilers only interpret identifier names with seven or fewer characters, the first seven characters of the FORTRAN-77 HDF routine names are unique.

1.5.4 Header Files

The inclusion of header files is not generally permitted by FORTRAN-77 compilers. However, it is sometimes available as an option. On UNIX systems, for example, the macro processors \texttt{m4} and \texttt{cpp} let the compiler include and preprocess header files. If this capability is not available, the user may have to copy the declarations, definitions, and values needed from the files \texttt{dffunc.inc} and \texttt{hdf.inc} into the user application. If the capability is available, the files can be included in the Fortran code. The files reside in the \texttt{include/} subdirectory of the directory where the HDF library is installed on the user’s system.

1.5.5 Data Type Specifications

When mixing machines, compilers, and languages, it is difficult to maintain consistent data type definitions. For instance, on some machines an integer is a 32-bit quantity and on others, a 16-bit quantity. In addition, the differences between FORTRAN-77 and C lead to difficulties in describing the data types found in the argument lists of HDF routines. To maintain portability, the HDF library expects assigned names for all data types used in HDF routines. (See Table 1A.)
When using a FORTRAN-77 data type that is not supported, the general practice is to use another data type of the same size. For example, an 8-bit signed integer can be used to store an 8-bit unsigned integer variable unless the code relies on a sign-specific operation.

### 1.5.6 String and Array Specifications

In the declarations contained in the headers of FORTRAN-77 functions, the following conventions are followed:

- `character(*) x` means that `x` refers to a string of an indefinite number of characters. It is the responsibility of the calling program to allocate enough space to hold the data to be stored in the string.
- `real x(*)` means that `x` refers to an array of reals of indefinite size and of indefinite rank. It is the responsibility of the calling program to allocate an actual array with the correct number of dimensions and dimension sizes.
- `<valid numeric data type> x` means that `x` may have one of the numeric data types listed in the Description column of Table 1A on page 6.
- `<valid data type> x` means that `x` may have any of the data types listed in the Description column of Table 1A on page 6.

### 1.5.7 FORTRAN-77, ANSI C and K&R C

As much as possible, we have conformed the HDF API routines to those implementations of FORTRAN and C that are in most common use today, namely FORTRAN-77, ANSI C and K&R C. Due to the increasing availability of ANSI C, future versions of HDF will no longer support K&R C.
As Fortran-90 is a superset of FORTRAN-77, HDF programs should compile and run correctly when using a Fortran-90 compiler.

### 1.6 Error Codes

The error codes defined in the HDF library are listed in the following table.

<table>
<thead>
<tr>
<th>HDF Error Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Error Code</strong></td>
</tr>
<tr>
<td>DFE_NONE</td>
</tr>
<tr>
<td>DFE_FNF</td>
</tr>
<tr>
<td>DFE_DENIED</td>
</tr>
<tr>
<td>DFE_ALROPen</td>
</tr>
<tr>
<td>DFE_TooMANY</td>
</tr>
<tr>
<td>DFE_BAdNAME</td>
</tr>
<tr>
<td>DFE_BAdACC</td>
</tr>
<tr>
<td>DFE_BAdOPEN</td>
</tr>
<tr>
<td>DFE_NOTOPEN</td>
</tr>
<tr>
<td>DFE_CANTCLOSE</td>
</tr>
<tr>
<td>DFE_READERROR</td>
</tr>
<tr>
<td>DFE_WRITEERROR</td>
</tr>
<tr>
<td>DFE_SEEKERROR</td>
</tr>
<tr>
<td>DFE_RDONLY</td>
</tr>
<tr>
<td>DFE_BADSEEK</td>
</tr>
<tr>
<td>DFE_PUTELEM</td>
</tr>
<tr>
<td>DFE_GETELEM</td>
</tr>
<tr>
<td>DFE_CantLink</td>
</tr>
<tr>
<td>DFE_CANTSYNC</td>
</tr>
<tr>
<td>DFE_BADGROUP</td>
</tr>
<tr>
<td>DFE_GROUPSETup</td>
</tr>
<tr>
<td>DFE_PUTCROUP</td>
</tr>
<tr>
<td>DFE_GROUPWRITE</td>
</tr>
<tr>
<td>DFE_DFNULL</td>
</tr>
<tr>
<td>DFE_ILLTYPE</td>
</tr>
<tr>
<td>DFE_BADDDLIST</td>
</tr>
<tr>
<td>DFE_NOTOFFILE</td>
</tr>
<tr>
<td>DFE_SEEDVICE</td>
</tr>
<tr>
<td>DFE_NOPUCHTAG</td>
</tr>
<tr>
<td>DFE_NOPFREEDD</td>
</tr>
<tr>
<td>DFE_BADTAG</td>
</tr>
<tr>
<td>DFE_BADREF</td>
</tr>
<tr>
<td>DFE_MOGATCH</td>
</tr>
<tr>
<td>DFE_NOTINSET</td>
</tr>
<tr>
<td>DFE_BADOFFSET</td>
</tr>
<tr>
<td>DFE_CORRUPT</td>
</tr>
<tr>
<td>DFE_MOREF</td>
</tr>
<tr>
<td>DFE_DUPOD</td>
</tr>
<tr>
<td>DFE_CANTMOD</td>
</tr>
<tr>
<td>Error Code</td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>DFE_DIFFFILES</td>
</tr>
<tr>
<td>DFE_BADAID</td>
</tr>
<tr>
<td>DFE_OPENAID</td>
</tr>
<tr>
<td>DFE_CANTFLUSH</td>
</tr>
<tr>
<td>DFE_CANTUPDATE</td>
</tr>
<tr>
<td>DFE_CANTHASH</td>
</tr>
<tr>
<td>DFE_CANTDELDD</td>
</tr>
<tr>
<td>DFE_CANTDELHASH</td>
</tr>
<tr>
<td>DFE_CANTACCESS</td>
</tr>
<tr>
<td>DFE_TABLEFULL</td>
</tr>
<tr>
<td>DFE_NOTINTABLE</td>
</tr>
<tr>
<td>DFE_UNSUPPORTED</td>
</tr>
<tr>
<td>DFE_NOSPACE</td>
</tr>
<tr>
<td>DFE_BADCALL</td>
</tr>
<tr>
<td>DFE_BADPTR</td>
</tr>
<tr>
<td>DFE_BADLEN</td>
</tr>
<tr>
<td>DFE_NOTENOUGH</td>
</tr>
<tr>
<td>DFE_NOVALS</td>
</tr>
<tr>
<td>DFE_ARGS</td>
</tr>
<tr>
<td>DFE_INTERNAL</td>
</tr>
<tr>
<td>DFE_MORESET</td>
</tr>
<tr>
<td>DFE_GENAPP</td>
</tr>
<tr>
<td>DFE_UNINIT</td>
</tr>
<tr>
<td>DFE_CANTINIT</td>
</tr>
<tr>
<td>DFE_CANTSHUTDOWN</td>
</tr>
<tr>
<td>DFE_BADDIM</td>
</tr>
<tr>
<td>DFE_BADFP</td>
</tr>
<tr>
<td>DFE_BADDATATYPE</td>
</tr>
<tr>
<td>DFE_BADMCTYPE</td>
</tr>
<tr>
<td>DFE_BADNOMTYPE</td>
</tr>
<tr>
<td>DFE_BADORDER</td>
</tr>
<tr>
<td>DFE_RANGE</td>
</tr>
<tr>
<td>DFE_BADCONV</td>
</tr>
<tr>
<td>DFE_BADTYPE</td>
</tr>
<tr>
<td>DFE_BADSCHEME</td>
</tr>
<tr>
<td>DFE_BADMODEL</td>
</tr>
<tr>
<td>DFE_BADCODER</td>
</tr>
<tr>
<td>DFE_MODEL</td>
</tr>
<tr>
<td>DFE_CODER</td>
</tr>
<tr>
<td>DFE_CINIT</td>
</tr>
<tr>
<td>DFE_CDECODE</td>
</tr>
<tr>
<td>DFE_CENCODE</td>
</tr>
<tr>
<td>DFE_CTERM</td>
</tr>
<tr>
<td>DFE_CSEEK</td>
</tr>
<tr>
<td>DFE_MINIT</td>
</tr>
<tr>
<td>DFE_COMPINFO</td>
</tr>
<tr>
<td>Error Code</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>DFE_CANTCOMP</td>
</tr>
<tr>
<td>DFE_CANTDECOMP</td>
</tr>
<tr>
<td>DFE_NODIM</td>
</tr>
<tr>
<td>DFE_BADRIG</td>
</tr>
<tr>
<td>DFE_RINOTFOUND</td>
</tr>
<tr>
<td>DFE_BADATTR</td>
</tr>
<tr>
<td>DFE.BADTABLE</td>
</tr>
<tr>
<td>DFE_BADSDG</td>
</tr>
<tr>
<td>DFE_BADNDG</td>
</tr>
<tr>
<td>DFE_VGSIZE</td>
</tr>
<tr>
<td>DFE_VTAB</td>
</tr>
<tr>
<td>DFE_CANTADDELEM</td>
</tr>
<tr>
<td>DFE_BADVGNAME</td>
</tr>
<tr>
<td>DFE_BADVGCLASS</td>
</tr>
<tr>
<td>DFE_BADFIELDS</td>
</tr>
<tr>
<td>DFE_NOVS</td>
</tr>
<tr>
<td>DFE_SYMSIZE</td>
</tr>
<tr>
<td>DFE_BADATTACH</td>
</tr>
<tr>
<td>DFE_BADVSNAME</td>
</tr>
<tr>
<td>DFE_BADVSCLASS</td>
</tr>
<tr>
<td>DFE_VSWRITE</td>
</tr>
<tr>
<td>DFE_VSREAD</td>
</tr>
<tr>
<td>DFE_BADVR</td>
</tr>
<tr>
<td>DFE_VSCANTCREATE</td>
</tr>
<tr>
<td>DFE_VOVCANTCREATE</td>
</tr>
<tr>
<td>DFE_CANTATTACH</td>
</tr>
<tr>
<td>DFE_CANTDETACH</td>
</tr>
<tr>
<td>DFE_BITREAD</td>
</tr>
<tr>
<td>DFE_BITWRITE</td>
</tr>
<tr>
<td>DFE_BITSEEK</td>
</tr>
<tr>
<td>DFE_TBBTINS</td>
</tr>
<tr>
<td>DFE_BVNEW</td>
</tr>
<tr>
<td>DFE_BVSET</td>
</tr>
<tr>
<td>DFE_BVGET</td>
</tr>
<tr>
<td>DFE_BVFIND</td>
</tr>
</tbody>
</table>
2.1 Reference Section Overview

This section of the Reference Manual contains a listing of every routine contained in the HDF version 4.1r4 library. For each interface, the pages are organized alphabetically according to the C routine name. Each page addresses one C routine and the related FORTRAN-77 routines, and takes the following form:

**Routine_Name**

\[ \text{return_type function_name} \left( \text{type1 parameter1}, \text{type2 parameter2}, \ldots, \text{typeN parameterN} \right) \]

- **parameter1**: IN/OUT: Definition of the first parameter
- **parameter2**: IN/OUT: Definition of the second parameter
- \ldots
- **parameterN**: IN/OUT: Definition of the Nth parameter

**Purpose**: Section containing the functionality of the routine.

**Return value**: Section describing the return value, if any.

**Description**: This optional section describes the proper use of the routine, the specification of the parameters, and any special circumstances surrounding the use of the routine. This section also identifies any prerequisite routines and provides appropriate references.

**FORTRAN**: This section provides a synopsis of the equivalent FORTRAN-77 routine or routines.
**ANannlen/afannlen**

`int32 ANannlen(int32 ann_id)`

*ann_id*  
IN: Annotation identifier returned by `ANcreate`, `ANcreatef`, or `ANselect`

**Purpose**
Returns the length of an annotation.

**Return value**
Returns the length of the annotation or `FAIL` (or `-1`) otherwise.

**Description**
`ANannlen` returns the number of characters contained in the annotation specified by the parameter `ann_id`. This function is commonly used to determine the size of a buffer to store the annotation upon reading.

**FORTRAN**

```fortran
integer function afannlen(ann_id)

integer ann_id
```
## ANannlist/afannlist

**Purpose**
Retrieves the annotation identifiers of an object.

**Return value**
Returns SUCCEED (or 0) or FAIL (or -1) otherwise.

**Description**
ANannlist obtains a list of identifiers of the annotations that are of the type specified by the parameter `annot_type` and are attached to the object identified by its tag, `obj_tag`, and its reference number, `obj_ref`.

Since this routine is implemented only to obtain the identifiers of data annotations and not file annotations, the valid values of `annot_type` are `AN_DATA_LABEL` (or 0) and `AN_DATA_DESC` (or 1). To obtain file annotation identifiers, use ANfileinfo to determine the number of file labels and descriptions, and then use ANselect to obtain each file annotation identifier.

Sufficient space must be allocated for `ann_list` to hold the list of annotation identifiers. This can be done by using ANnumann to obtain the number of annotation identifiers to be retrieved, and then allocating memory for `ann_list` using this number.

### FORTRAN

```fortran
integer function afannlist(an_id, annot_type, obj_tag, obj_ref, ann_list)

integer ann_list(*)

integer an_id, obj_tag, obj_ref, annot_type
```

### Example C function

```c
int ANannlist(int32 an_id, uint16 annot_type, uint16 obj_tag, uint16 obj_ref, int32 *ann_list)

an_id IN: AN interface identifier returned by ANstart
annot_type IN: Type of the annotation
obj_tag IN: Tag of the object
obj_ref IN: Reference number of the object
ann_list OUT: Buffer for the annotation identifiers
```
ANatype2tag/afatypetag

uint16 ANatype2tag(ann_type *annot_type)

annot_type IN: Type of the annotation

Purpose
Returns the annotation tag corresponding to an annotation type.

Return value
Returns the annotation tag (ann_tag) if successful, and DFTAG_NULL (or 0) otherwise.

Description
ANatype2tag returns the tag that corresponds to the annotation type specified by the parameter annot_type.

The following table lists the valid values of annot_type in the left column and the corresponding values for the returned annotation tag on the right.

<table>
<thead>
<tr>
<th>Annotation Type</th>
<th>Annotation Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>AN_DATA_LABEL (or 0)</td>
<td>DFTAG_DIL (or 104)</td>
</tr>
<tr>
<td>AN_DATA_DESC (or 1)</td>
<td>DFTAG_DIA (or 105)</td>
</tr>
<tr>
<td>AN_FILE_LABEL (or 2)</td>
<td>DFTAG_FID (or 100)</td>
</tr>
<tr>
<td>AN_FILE_DESC (or 3)</td>
<td>DFTAG_FD (or 101)</td>
</tr>
</tbody>
</table>

FORTRAN
integer function afatypetag(annot_type)

integer annot_type
**ANcreate/afcreate**

**Purpose**
Creates a data annotation for an object.

**Return value**
Returns the data annotation identifier (ann_id) if successful and \texttt{FAIL} (or -1) otherwise.

**Description**
\texttt{ANcreate} creates a data annotation of type \texttt{annot_type} for the object specified by its tag, \texttt{obj_tag}, and its reference number, \texttt{obj_ref}. The returned data annotation identifier can represent either a data label or a data description.

Valid values for \texttt{annot_type} are \texttt{AN\_DATA\_LABEL} (or 0) or \texttt{AN\_DATA\_DESC} (or 1).

Use \texttt{ANcreatef} 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 \texttt{ANcreate} to return \texttt{FAIL} (or -1).

**FORTRAN**

```fortran
integer function afcreate(an_id, obj_tag, obj_ref, annot_type)
    integer an_id, obj_tag, obj_ref, annot_type
```

```c
int32 ANcreate(int32 an_id, uint16 obj_tag, uint16 obj_ref, ann_type annot_type)
```

\begin{itemize}
  \item \texttt{an_id} \textbf{IN:} AN interface identifier returned by \texttt{ANstart}
  \item \texttt{obj_tag} \textbf{IN:} Tag of the object to be annotated
  \item \texttt{obj_ref} \textbf{IN:} Reference number of the object to be annotated
  \item \texttt{annot_type} \textbf{IN:} Type of the data annotation
\end{itemize}
ANcreatef/affcreate

int32 ANcreatef(int32 an_id, ann_type annot_type)

  an_id  IN: AN interface identifier returned by ANstart
  annot_type  IN: Type of the file annotation

Purpose
 Creates a file annotation.

Return value
 Returns the file annotation identifier (ann_id) if successful and FAIL (or -1) otherwise.

Description
 ANcreatef creates a file annotation of the type specified by the parameter annot_type. The file annotation identifier returned can either represent a file label or a file description.

 Valid values for annot_type are AN_FILE_LABEL (or 2) and AN_FILE_DESC (or 3).

 Use ANcreate 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 ANcreate to return FAIL (or -1).

FORTRAN

 integer function affcreate(an_id, annot_type)

 integer an_id, annot_type
ANend/afend

int32 ANend(int32 an_id)

\[ an_id \quad \text{IN: AN interface identifier returned by ANstart} \]

Purpose
Terminates access to an AN interface.

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

Description
ANend terminates access to the AN interface identified by \( an_id \), which is previously initialized by a call to ANstart. Note that there must be one call to ANend for each call to ANstart.

FORTRAN
integer function afend(an_id)

integer an_id
ANEndaccess/afendaccess

int ANEndaccess(int32 ann_id)

ann_id IN: Annotation identifier returned by ANcreate, ANcreatef or ANselect

Purpose Terminates access to an annotation.

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

Description ANEndaccess terminates access to the annotation identified by the parameter ann_id. Note that there must be one call to ANEndaccess for every call to ANselect, ANcreate or ANcreatef.

FORTRAN integer function afendaccess(ann_id)

integer ann_id
ANfileinfo

ANfileinfo/retrieveinfo

intn ANfileinfo(int32 an_id, int32 *n_file_labels, int32 *n_file_descs, int32 *n_data_labels, int32 *n_data_descs)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>an_id</td>
<td>IN: AN interface identifier returned by ANstart</td>
</tr>
<tr>
<td>n_file_labels</td>
<td>OUT: Number of file labels</td>
</tr>
<tr>
<td>n_file_descs</td>
<td>OUT: Number of file descriptions</td>
</tr>
<tr>
<td>n_data_labels</td>
<td>OUT: Number of data labels</td>
</tr>
<tr>
<td>n_data_descs</td>
<td>OUT: Number of data descriptions</td>
</tr>
</tbody>
</table>

**Purpose**
Retrieves the number of annotations of each type in a file.

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

**Description**
ANfileinfo retrieves the total number of the four kinds of annotations and stores them in the appropriate parameters. The total number of data labels of all data objects in the file is stored in n_data_labels. The total number of data descriptions of all data objects in the file is stored in n_data_descs. The total number of file labels is stored in n_file_labels and the total number of file descriptions in n_file_descs.

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 ANnumann to determine these numbers for a specific object.

This routine is generally used to find the range of acceptable indices for ANselect calls.

**FORTRAN**

integer function ANfileinfo(an_id, n_file_labels, n_file_descs, n_data_labels, n_data_descs)

integer an_id, n_file_labels, n_file_descs

integer n_data_labels, n_data_descs
ANget_tagref/afgettagref

int32 ANget_tagref(int32 an_id, int32 index, int_type annot_type, uint16 *ann_tag, uint16 *ann_ref)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>an_id</td>
<td>AN interface identifier returned by ANstart</td>
</tr>
<tr>
<td>index</td>
<td>Index of the annotation</td>
</tr>
<tr>
<td>annot_type</td>
<td>Type of the annotation</td>
</tr>
<tr>
<td>ann_tag</td>
<td>Tag of the annotation</td>
</tr>
<tr>
<td>ann_ref</td>
<td>Reference number of the annotation</td>
</tr>
</tbody>
</table>

**Purpose**
Retrieves the tag/reference number pair of an annotation given its index and type.

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

**Description**
ANget_tagref retrieves the tag and reference number of the annotation identified by its index, the parameter index, and by its annotation type, the parameter annot_type. The tag is stored in the parameter ann_tag and the reference number is stored in the parameter ann_ref.

The parameter index is a nonnegative integer and is less than the total number of annotations of type annot_type in the file. Use ANfileinfo to obtain the total number of annotations of each type in the file.

The following table lists the valid values of the parameter annot_type in the left column, and the corresponding values of the parameter ann_tag in the right column.

<table>
<thead>
<tr>
<th>Annotation Type</th>
<th>Annotation Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>AN_DATA_LABEL (or 0)</td>
<td>DFTAG_DIL (or 104)</td>
</tr>
<tr>
<td>AN_DATA_DESC (or 1)</td>
<td>DFTAG_DIA (or 105)</td>
</tr>
<tr>
<td>AN_FILE_LABEL (or 2)</td>
<td>DFTAG_FID (or 100)</td>
</tr>
<tr>
<td>AN_FILE_DESC (or 3)</td>
<td>DFTAG_FD (or 101)</td>
</tr>
</tbody>
</table>

**FORTRAN**
integer function afgettagref(an_id, index, annot_type, ann_tag, ann_ref)

integer an_id, index, annot_type

integer ann_tag, ann_ref
ANid2tagref/afidtagref

int32 ANid2tagref(int32 ann_id, uint16 *ann_tag, uint16 *ann_ref)

ann_id       IN:   Annotation identifier returned by ANselect, ANcreate or ANcreatef
ann_tag       OUT:  Tag of the annotation
ann_ref       OUT:  Reference number of the annotation

Purpose    Retrieves the tag/reference number pair of an annotation given its identifier.
Return value  Returns SUCCEED (or 0) if successful or FAIL (or -1) otherwise.
Description ANid2tagref retrieves the tag/reference number pair of the annotation identified by the parameter ann_id. The tag is stored in the parameter ann_tag and the reference number is stored in the parameter ann_ref.

Possible values returned in ann_tag are DFTAG_DIL (or 104) for a data label, DFTAG_DIA (or 105) for a data description, DFTAG_FID (or 100) for a file label and DFTAG_FD (or 101) for a file description.

FORTRAN    integer function afidtagref(ann_id, ann_tag, ann_ref)

integer ann_id, ann_tag, ann_ref
ANnumann/afnumann

intn ANnumann(int32 an_id, ann_type annot_type, uint16 obj_tag, uint16 obj_ref)

\begin{tabular}{ll}
\textit{an_id} & IN: AN interface identifier returned by \texttt{ANstart} \\
\textit{annot_type} & IN: Type of the annotation \\
\textit{obj_tag} & IN: Tag of the object \\
\textit{obj_ref} & IN: Reference number of the object \\
\end{tabular}

\textbf{Purpose} \hspace{1cm} Returns the number of annotations of a given type attached to an object.

\textbf{Return value} \hspace{1cm} Returns the number of annotations or \texttt{FAIL} (or -1) otherwise.

\textbf{Description} \hspace{1cm} ANnumann returns the total number of annotations that are of type \texttt{annot_type} and that are attached to the object identified by its tag, \texttt{obj_tag}, and its reference number, \texttt{obj_ref}.

Since this routine is implemented only to obtain the total number of data annotations and not file annotations, the valid values of \texttt{annot_type} are \texttt{AN_DATA_LABEL} (or 0) and \texttt{AN_DATA_DESC} (or 1). To obtain the total number of file annotations or all data annotations, use \texttt{ANfileinfo}.

\textbf{FORTRAN} \hspace{1cm} integer function afnumann(an_id, annot_type, obj_tag, obj_ref)

integer an_id, obj_tag, obj_ref, annot_type
**ANreadann/afreadann**

int32 ANreadann(int32 ann_id, char* ann_buf, int32 ann_length)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type/Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ann_id</td>
<td>IN: Annotation identifier returned by <strong>ANcreate, ANcreatef</strong> or <strong>ANselect</strong></td>
</tr>
<tr>
<td>ann_buf</td>
<td>OUT: Buffer for the annotation</td>
</tr>
<tr>
<td>ann_length</td>
<td>IN: Length of the buffer ann_buf</td>
</tr>
</tbody>
</table>

**Purpose**
Reads an annotation.

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

**Description**
**ANreadann** reads the annotation identified by the parameter **ann_id** and stores the annotation in the parameter **ann_buf**.

The parameter **ann_length** specifies the size of the buffer **ann_buf**. If the length of the file or data label to be read is greater than or equal to **ann_length**, the label will be truncated to **ann_length - 1** characters. If the length of the file or data description is greater than **ann_length**, the description will be truncated to **ann_length** characters. The HDF library adds a **NULL** character to the retrieved label but not to the retrieved description. The user must add a **NULL** character to the retrieved description if the C library string functions are to operate on this description.

**FORTRAN**

```
integer function afreadann(ann_id, ann_buf, ann_length)

integer ann_id, ann_length
character(*) ann_buf
```
ANselect/afselect

int32 ANselect(int32 an_id, int32 index, ann_type annot_type)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>an_id</td>
<td>IN: AN interface identifier returned by ANstart</td>
</tr>
<tr>
<td>index</td>
<td>IN: Location of the annotation in the file</td>
</tr>
<tr>
<td>annot_type</td>
<td>IN: Type of the annotation</td>
</tr>
</tbody>
</table>

Purpose
Obtains an existing annotation.

Return value
Returns the annotation identifier (ann_id) if successful or FAIL (or -1) otherwise.

Description
ANselect obtains the identifier of the annotation specified by its index, index, and by its annotation type, annot_type.

The parameter index is a nonnegative integer and is less than the total number of annotations of type annot_type in the file. Use ANfileinfo to obtain the total number of annotations of each type in the file.

Valid values of annot_type are AN_DATA_LABEL (or 0), AN_DATA_DESC (or 1), AN_FILE_LABEL (or 2), and AN_FILE_DESC (or 3).

FORTRAN
integer function afselect(an_id, index, annot_type)

integer an_id, index
integer annot_type
ANstart/afstart

int32 ANstart(int32 file_id)

  file_id  IN:  File identifier returned by Hopen

Purpose  Initializes the AN interface.
Return value  Returns the AN interface identifier (an_id) if successful and FAIL (or -1) otherwise.
Description  ANstart initializes the AN interface for the file identified by the parameter file_id. A call to ANstart is required before any AN functions can be invoked. ANstart is used with the ANend function to define the extent of AN interface session. A call to ANend is required for each call to ANstart.

FORTRAN  integer function afstart(file_id)

    integer file_id
ann_type ANtag2atype(uint16 ann_tag)

ann_tag IN: Tag of the annotation

Purpose Returns the annotation type corresponding to an annotation tag.

Return value Returns the annotation type if successful or AN_UNDEF (or -1) otherwise.

Description ANtag2atype returns the annotation type that corresponds to the annotation tag specified by the parameter ann_tag.

The following table lists the valid values of ann_tag in the left column and the corresponding values of the returned annotation type in the right column.

<table>
<thead>
<tr>
<th>Annotation Tag</th>
<th>Annotation Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFTAG_DIL (or 104)</td>
<td>AN_DATA_LABEL (or 0)</td>
</tr>
<tr>
<td>DFTAG_DIA (or 105)</td>
<td>AN_DATA_DESC (or 1)</td>
</tr>
<tr>
<td>DFTAG_FID (or 100)</td>
<td>AN_FILE_LABEL (or 2)</td>
</tr>
<tr>
<td>DFTAG_FD (or 101)</td>
<td>AN_FILE_DESC (or 3)</td>
</tr>
</tbody>
</table>

FORTRAN integer function aftagatype(ann_tag)

integer ann_tag
ANtagref2id/aftagrefid

int32 ANtagref2id(int32 an_id, uint16 ann_tag, uint16 ann_ref)

an_id IN: AN interface identifier returned by ANstart
ann_tag IN: Tag of the annotation
ann_ref IN: Reference number of the annotation

Purpose
Returns the identifier of an annotation given its tag/reference number pair.

Return value
Returns the annotation identifier (ann_id) if successful and FAIL (or -1) otherwise.

Description
ANtagref2id returns the identifier of the annotation specified by its tag, ann_tag, and its reference number, ann_ref.

Valid values of ann_tag are DFTAG_DIL (or 104) for a data label, DFTAG_DIA (or 105) for a data description, DFTAG_FID (or 100) for a file label, and DFTAG_FD (or 101) for a file description.

FORTRAN
integer function aftagrefid(an_id, ann_tag, ann_ref)

integer an_id, ann_tag, ann_ref
ANwriteann/afwriteann

int32 ANwriteann(int32 ann_id, char* ann, int32 ann_length)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ann_id</td>
<td>Annotation identifier returned by ANcreate, ANcreatef, or ANselect</td>
</tr>
<tr>
<td>ann</td>
<td>Text to be written to the annotation</td>
</tr>
<tr>
<td>ann_length</td>
<td>Length of the annotation text</td>
</tr>
</tbody>
</table>

**Purpose**

Writes an annotation.

**Return value**

Returns **SUCCEED** (or 0) if successful and **FAIL** (or -1) otherwise.

**Description**

**ANwriteann** writes the annotation text provided in the parameter *ann* to the annotation specified by the parameter *ann_id*. The parameter *ann_length* specifies the number of characters in the annotation text.

If the annotation has already been written with text, **ANwriteann** will overwrite the current text.

**FORTRAN**

```fortran
integer function afwriteann(ann_id, ann, ann_length)
integer ann_id, ann_length
character(*) ann
```
GRattrinfo/mgatinf

intn GRattrinfo(int32 [obj]_id, int32 attr_index, char *name, int32 *data_type, int32 *count)

{obj]_id IN: Raster image identifier (ri_id), returned by GRcreate or GRselect, or GR interface identifier (gr_id), returned by GRstart
attr_index IN: Index of the attribute
name OUT: Buffer for the name of the attribute
data_type OUT: Data type of the attribute
count OUT: Number of attribute values

Purpose Retrieves information about an attribute.
Return value Returns succeed (or 0) if successful and fail (or -1) otherwise.
Description GRattrinfo retrieves the name, data type, and number of values of the attribute, specified by its index, attr_index, for the data object identified by the parameter obj_id. The name is stored in the parameter name, the data type is stored in the parameter data_type, and the number of values is stored in the parameter count. If the value of any of the output parameters is NULL, the corresponding information will not be retrieved.

The value of the parameter attr_index can be obtained using GRfindattr, GRnametoindex or GRreftoindex, depending on available information. Valid values of attr_index range from 0 to the total number of attributes attached to the object - 1. The total number of attributes attached to the file can be obtained using the routine GRfileinfo. The total number of attributes attached to an image can be obtained using the routine GRgetiminfo.

FORTRAN integer function mgatinf([obj]_id, attr_index, name, data_type, count)

ingteger [obj]_id, data_type, attr_index, count
character*(*) name
GRcreate/mgcreate

GRcreate/mgcreate

int32 GRcreate(int32 gr_id, char *name, int32 ncomp, int32 data_type, int32 interlace_mode, int32 dim_sizes[2])

gr_id IN: GR interface identifier returned by GRstart
name IN: Name of the raster image
ncomp IN: Number of pixel components in the image
data_type IN: Type of the image data
interlace_mode IN: Interlace mode of the image data
dim_sizes IN: Size of each dimension of the image

Purpose
Creates a new raster image.

Return value
Returns a raster image identifier if successful and FAIL (or -1) otherwise.

Description
GRcreate creates a raster image with the values provided in the parameters name, ncomp, data_type, interlace_mode and dim_sizes.

The parameter name specifies the name of the image and must not be NULL. The length of the name should not be longer than MAX_GR_NAME (or 256).

The parameter ncomp specifies the number of pixel components in the raster image and must have a value of at least 1.

The parameter data_type specifies the type of the raster image data and can be any of the data types supported by the HDF library. The data types supported by HDF are listed in Table 1A in Section I of this manual.

The parameter interlace_mode specifies the interlacing in which the raster image is to be written. The valid values of interlace_mode are: MFGR_INTERLACE_PIXEL (or 0), MFGR_INTERLACE_LINE (or 1) and MFGR_INTERLACE_COMPONENT (or 2).

The array dim_sizes specifies the size of the two dimensions of the image. The dimensions must be specified and their values must be greater than 0.

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 GRselect.

Images created with the GR interface are actually written to disk in pixel interlace mode; any user-specified interlace mode is stored in the file with the image and the image is automatically converted to that mode when it is read with a GR interface function.
Note

Regarding an important difference between the SD and GR interfaces:
The SD and GR interfaces differ in the correspondence between the dimension order in parameter arrays such as `start`, `stride`, `edge`, and `dimsizes` and the dimension order in the `data` array. See the `SDreaddata` and `GRreadimage` reference manual pages for discussions of the SD and GR approaches, respectively.

When writing applications or tools to manipulate both images and two-dimensional SDs, this crucial difference between the interfaces must be taken into account. While the underlying data is stored in row-major order in both cases, the API parameters are not expressed in the same way. Consider the example of an SD data set and GR image that are stored as identically-shaped arrays of X columns by Y rows and accessed via the `SDreaddata` and `GRreadimage` functions, respectively. Both functions take the parameters `start`, `stride`, and `edge`:

- For `SDreaddata`, those parameters are expressed in (y,x) or [row,column] order. For example, `start[0]` is the starting point in the Y dimension and `start[1]` is the starting point in the X dimension. The same ordering holds true for all SD data set manipulation functions.
- For `GRreadimage`, those parameters are expressed in (x,y) or [column,row] order. For example, `start[0]` is the starting point in the X dimension and `start[1]` is the starting point in the Y dimension. The same ordering holds true for all GR functions manipulating image data.

FORTRAN

```fortran
integer function mgcreat(gr_id, name, ncomp, data_type,
                        interlace_mode, dim_sizes)

integer gr_id, data_type, interlace_mode, ncomp, dim_sizes(2)
character(*) name
```
intn GRend(int32 gr_id)

**Purpose**
Terminates the GR interface session.

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

**Description**
**GRend** terminates the GR interface session identified by the parameter *gr_id*.

**GRend**, together with **GRstart**, defines the extent of a GR interface session. **GRend** disposes of the internal structures initialized by the corresponding call to **GRstart**. There must be a call to **GRend** for each call to **GRstart**; failing to provide one may cause loss of data.

**GRstart** and **GRend** do not manage file access; use **Hopen** and **Hclose** to open and close HDF files. **Hopen** must be called before **GRstart** and **Hclose** must be called after **GRend**.

**FORTRAN**

```fortran
integer function mgend(gr_id)
    integer gr_id
```

intn GRendaccess(int32 ri_id)

*ri_id*  IN:  Raster image identifier returned by *GRcreate* or *GRselect*

**Purpose**
Terminates access to a raster image.

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

**Description**
*GRendaccess* 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 *GRselect* or *GRcreate*. There must be a call to *GRendaccess* for each call to *GRselect* or *GRcreate*; failing to provide this will result in loss of data. Attempts to access a raster image identifier disposed of by *GRendaccess* will result in an error condition.

**FORTRAN**

```
integer function mgendac(ri_id)

integer ri_id
```
GRfileinfo

GRfileinfo/GRfileinfo

intn GRfileinfo(int32 gr_id, int32 *n_images, int32 *n_file_attrs)

gr_id   IN:   GR interface identifier returned by GRstart
n_images OUT: Number of raster images in the file
n_file_attrs OUT: Number of global attributes in the file

Purpose
Retrieves the number of raster images and the number of global attributes in the file.

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

Description
GRfileinfo retrieves the number of raster images and the number of global attributes for the 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 GRsetattr with the object identifier parameter set to a GR interface identifier (gr_id) rather than a raster image identifier (ri_id).

GRfileinfo is useful in finding the range of acceptable indices for GRselect calls.

FORTRAN

integer function mgfinfo(gr_id, n_images, n_file_attrs)

integer gr_id, n_images, n_file_attrs
GRfindattr/mgfndat

int32 GRfindattr(int32 \[obj\]_id, char *\attr\_name\)

\[obj\]_id  IN:  Raster image identifier (ri_id), returned by GRcreate or GRselect, or GR interface identifier (gr_id), returned by GRstart
\attr\_name  IN:  Name of the attribute

Purpose  Finds the index of a data object’s attribute given an attribute name.

Return value  Returns the index of the attribute if successful and FAIL (or -1) otherwise.

Description  \texttt{GRfindattr} returns the index of the attribute whose name is specified by the parameter \texttt{attr\_name} for the object identified by the parameter \texttt{obj\_id}.

FORTRAN  integer function mgfndat\([\texttt{obj\_id}}, \attr\_name\)

\texttt{integer} \texttt{[obj]}_id

\texttt{character\{\texttt{*}\}} \attr\_name
GRgetattr/mggnatt/mggcatt

intn GRgetattr(int32 [obj]_id, int32 attr_index, VOIDP values)

[obj]_id IN: Raster image identifier (ri_id), returned by GRcreate or GRselect, or GR interface identifier (gr_id), returned by GRstart

attr_index IN: Index of the attribute

values OUT: Buffer for the attribute values

Purpose Reads the values of an attribute for a data object.

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

Description GRgetattr obtains all values of the attribute that is specified by its index, attr_index, and is attached to the object identified by the parameter obj_id. The values are stored in the buffer values.

The value of the parameter attr_index can be obtained by using GRfindattr, GRnametoindex, or GRreftoindex, 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 GRfileinfo. The total number of attributes attached to the image can be obtained using the routine GRgetiminfo.

GRgetattr only reads all values assigned to the attribute and not a subset.

Note that there are two FORTRAN-77 versions of this routine; one for numeric data (mggnatt) and the other for character data (mggcatt).

FORTRAN

integer function mggnatt([obj]_id, attr_index, values)

integer [obj]_id, attr_index
<valid numeric data type> values(*)

integer function mggcatt([obj]_id, attr_index, values)

integer [obj]_id, attr_index
character*(*) values
GRgetchunkinfo/mggichnk

intn GRgetchunkinfo(int32 ri_id, HDF_CHUNK_DEF *cdef, int32 *flag)

ri_id   IN:    Raster image identifier returned by GRcreate or GRselect

C only:
cdef   OUT:  Pointer to the chunk definition
flag    OUT:  Pointer to the compression flag

Fortran only:
dim_length  OUT:  Array of chunk dimensions
flag         OUT:  Compression flag

Purpose
Retrieves chunking information for a raster image.

Return value
Returns succeed (or 0) if successful and fail (or -1) otherwise.

Description
GRgetchunkinfo retrieves chunking information about the raster image
identified by the parameter ri_id into the parameters cdef and flags in C, and
into the parameters dim_length and flag in Fortran. Note that only chunk
dimensions are retrieved, compression information is not available.

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.

<table>
<thead>
<tr>
<th>Values of flag in C</th>
<th>Values of flag in Fortran</th>
<th>Raster Image Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDF_NONE</td>
<td>-1</td>
<td>Not chunked</td>
</tr>
<tr>
<td>HDF_CHUNK</td>
<td>0</td>
<td>Chunked and not compressed</td>
</tr>
<tr>
<td>HDF_CHUNK</td>
<td>HDF_COMP</td>
<td>1</td>
</tr>
</tbody>
</table>

In C, if the raster image is chunked and not compressed, GRgetchunkinfo fills
the array chunk_lengths in the union cdef with the values of the corresponding chunk dimensions. If the raster image is chunked and compressed, GRgetchunkinfo fills the array chunk_lengths in the structure comp of the union cdef with the values of the corresponding chunk dimensions. Refer to the page on GRsetchunk in this manual for specific information on the union HDF CHUNK DEF. In Fortran, chunk dimensions are retrieved into the array dim_length. If the chunk length for each dimension is not needed, NULL can be passed in as the value of the parameter cdef in C.
FORTRAN

integer function mggichnk(ri_id, dim_length, flag)

integer ri_id, dim_length, flag
**GRgetcompress/mggcompress**

```c
intn GRgetcompress(int32 ri_id, compcoder_t *comp_type, comp_info *c_info)
```

- `ri_id` **IN:** Raster image identifier returned by `GRcreate` or `GRselect`
- `comp_type` **OUT:** Type of compression

**C only:**
- `c_info` **OUT:** Pointer to compression information structure

**Fortran only:**
- `comp_prm` **OUT:** Compression parameters array

**Purpose**
Retrieves raster image data compression type and compression information.

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

**Description**
`GRgetcompress` retrieves the compression type and compression information for the specified raster image.

The compression method is returned in the parameter `comp_type`. Valid values of `comp_type` are as follows:

- `COMP_CODE_NONE` (or 0) for no compression
- `COMP_CODE_RLE` (or 1) for RLE run-length encoding
- `COMP_CODE_SKPHUFF` (or 3) for Skipping Huffman compression
- `COMP_CODE_DEFLATE` (or 4) for GZIP compression
- `COMP_CODE_SZIP` (or 5) for Szip compression
- `COMP_CODE_JPEG` (or 7) for JPEG compression

When a compression method requires additional parameters, those values are returned in the `c_info` struct in C and the array parameter `comp_prm` in Fortran.

The `c_info` struct is of type `comp_info`, contains algorithm-specific information for the library compression routines, and is described in the `hcomp.h` header file and in the `GRsetcompress` entry in this reference manual.

The `comp_prm` parameter is an array of one element:
- With Skipping Huffman compression, `comp_prm(1)` contains the skip value, `skphuff_skp_size`.
- In the case of GZIP compression, `comp_prm(1)` contains the deflation value, `deflate_value`.
- `comp_prm` is ignored with other compression methods. (There are no relevant RLE parameters and the `quality` and `force_baseline` data are not available for JPEG images. If `GRgetcompress` is called for either an RLE or a JPEG image, the function will return only the compression type; `c_info` will contain only zeros.)

**FORTRAN**

```fortran
integer function mggcompress(ri_id, comp_type, comp_prm)

integer ri_id, comp_type, comp_prm(1)
```
intn GRgetiminfo(int32 ri_id, char *gr_name, int32 *ncomp, int32 *data_type, int32 *interlace_mode, int32 *dim_sizes, int32 *num_attrs)

**ri_id** IN: Raster image identifier returned by **GRcreate** or **GRselect**

**gr_name** OUT: Buffer for the name of the raster image

**ncomp** OUT: Number of components in the raster image

**data_type** OUT: Data type of the raster image data

**interlace_mode** OUT: Interlace mode of the stored raster image data

**dim_sizes** OUT: Sizes of raster image dimension

**num_attrs** OUT: Number of attributes attached to the raster image

**Purpose** Retrieves general information about a raster image.

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

**Description** **GRgetiminfo** 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**.

**GRgetiminfo** stores the name, number of components, data type, interlace mode and dimension sizes of the image in the parameters **gr_name**, **ncomp**, **data_type**, **interlace_mode**, and **dim_sizes**, respectively. It also retrieves the number of attributes attached to the image into the parameter **num_attrs**. If the value of any of the output parameters are set to **NULL** in C, the corresponding information will not be retrieved.

The buffer **gr_name** is assumed to have sufficient space allocated to store the entire name of the raster image.

The valid values of the parameter **data_type** are listed in Table 1A in Section I of this manual.

**FORTRAN**

```
integer function mggiinf(ri_id, gr_name, ncomp, data_type, interlace_mode, dim_sizes, num_attrs)

integer ri_id, ncomp, data_type, interlace_mode, num_attrs
integer dim_sizes[2]
character*(*) gr_name
```
GRgetlutid/mggltid

int32 GRgetlutid(int32 ri_id, int32 pal_index)

ri_id       IN:  Raster image identifier returned by GRcreate or GRselect
pal_index   IN:  Index of the palette

Purpose    Gets the identifier of a palette given its index.
Return value Returns the palette identifier if successful and FAIL (or -1) otherwise.
Description GRgetlutid gets the identifier of the palette attached to the raster image identified by the parameter ri_id. The palette is identified by its index, pal_index.

Currently, only one palette can be assigned to a raster image, which means that pal_index should always be set to 0.

FORTRAN    integer function mggltid(ri_id, pal_index)

integer ri_id, pal_index
**GRgetlutinfo/mgglinf**

intn GRgetlutinfo(int32 pal_id, int32 *ncomp, int32 *data_type, int32 *interlace_mode, int32 *num_entries)

- **pal_id** IN: Palette identifier returned by GRgetlutid
- **ncomp** OUT: Number of components in the palette
- **data_type** OUT: Data type of the palette
- **interlace_mode** OUT: Interlace mode of the stored palette data
- **num_entries** OUT: Number of color lookup table entries in the palette

**Purpose**
Retrieves information about a palette.

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

**Description**
GRgetlutinfo 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*. These values are stored in the parameters *ncomp*, *data_type*, *interlace_mode*, and *num_entries*, respectively. In C if the value of any of the output parameters are set to NULL, the corresponding information will not be retrieved.

**FORTRAN**

```fortran
integer function mgglinf(pal_id, ncomp, data_type, interlace_mode, num_entries)

integer pal_id, ncomp, data_type, interlace_mode, num_entries
```
GRgetnluts/mggnluts

```c
intn GRgetnluts(int32 ri_id)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ri_id</code></td>
<td>IN: Data set identifier returned by <code>GRcreate</code> or <code>GRselect</code></td>
</tr>
</tbody>
</table>

**Purpose**
Retrieves the number of palettes for an image.

**Return value**
Returns number of palettes (1 or 0) if successful and `FAIL` (or -1) otherwise.

**Description**
`GRgetnluts` retrieves the number of palettes (or color look-up tables, commonly abbreviated as LUTs) available for the specified raster image.

There can currently be either 0 or 1 palettes assigned to an image. If multiple palettes are supported in a future release, this function may return values greater than 1.

**FORTRAN**

```fortran
integer function mggnluts(ri_id)
```

```fortran
integer ri_id
```
GRidtoref/mgid2rf

uint16 GRidtoref(int32 ri_id)

  ri_id  IN:  Raster image identifier returned by GRselect or GRcreate

Purpose  Maps a raster image identifier to a reference number.

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

Description  GRidtoref returns the 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 a vgroup.

FORTRAN

  integer function mgid2rf(ri_id)

  integer ri_id
GRlutteoref/mglt2rf

uint16 GRlutteoref(int32 pal_id)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pal_id</td>
<td>int32</td>
<td>Palette identifier returned by GRgetlutid</td>
</tr>
</tbody>
</table>

Purpose
Maps a palette identifier to a reference number.

Return value
Returns the reference number of the palette if successful or 0 otherwise.

Description
GRlutteoref returns the reference number of the palette identified by the parameter pal_id.

This routine is commonly used for the purpose of annotating the palette or including the palette within a vgroup.

FORTRAN

integer function mglt2rf(pal_id)

integer pal_id
int32 GRnametoindex(int32 gr_id, char *gr_name)

\[\text{Purpose}\]
Maps the name of a raster image to an index.

\[\text{Return value}\]
Returns the index of the raster image if successful and FAIL (or -1) otherwise.

\[\text{Description}\]
GRnametoindex returns, for the GR interface identified by the parameter \(gr\_id\), the index (\(index\)) of the raster image named \(gr\_name\).

The value of \(index\) can be passed into GRselect to obtain the raster image identifier (\(ri\_id\)).

\[\text{FORTRAN}\]
integer function mgn2ndx(gr_id, gr_name)

integer gr_id

character*(*) gr_name
GRreadchunk/mgrchnk/mgrcchnk

intn GRreadchunk(int32 ri_id, int32 *origin, VOIDP datap)

ri_id
IN: Raster image identiﬁer returned by GRcreate or GRselect

origin
IN: Origin of the chunk to be read

datap
IN: Buffer for the chunk to be read

Purpose
Reads a data chunk from a chunked raster image (pixel-interlace only)

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

Description
GRreadchunk reads the entire chunk of data from the chunked raster image identified
by the parameter ri_id and stores it in the buffer datap. Chunk to be read is
speciﬁed by the parameter origin. This function has less overhead than GRreadimage
and should be used whenever an entire chunk of data is to be read.

GRreadchunk will return FAIL (or -1) when an attempt is made to use it to
read from a non-chunked raster image.

The parameter origin is a two-dimensional array which speciﬁes the
coordinates of the chunk according to the chunk position in the overall chunk
array. Refer to Chapter 8, "General Raster Images (GR API)," in the HDF
User’s Guide.

The buffer datap contains the chunk data organized in pixel interlace mode.

FORTRAN

integer mgrchnk(ri_id, origin, datap)

integer ri_id, origin(2)
<valid_numeric_datatype> datap(*)

integer mgrcchnk(ri_id, origin, char_datap)

integer ri_id, origin(2)
character*(*) char_datap
GRreadimage reads the subsample of the raster image specified by the parameter ri_id into the buffer data. 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 read. Valid values of each element in the array start are 0 to the size of the corresponding raster image dimension - 1. The first element of the array start specifies an offset from the beginning of the array data along the fastest-changing dimension, which is the second dimension in C and the first dimension in Fortran. The second element of the array start specifies an offset from the beginning of the array data along the second fastest-changing dimension, which is the first dimension in C and the second dimension in Fortran. For example, if the first value of the array start is 2 and the second value is 3, the starting location of the subsample to be read is at the fourth row and third column in C, and at the third row and fourth column in Fortran.

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.

Note that there are two FORTRAN-77 versions of this routine; one for numeric data (mgrdimg) and the other for character data (mgrcimg).
Note

Regarding an important difference between the SD and GR interfaces:
The SD and GR interfaces differ in the correspondence between the dimension order in parameter arrays such as start, stride, edge, and dimsizes and the dimension order in the data array. See the SDreaddata and GRreadimage reference manual pages for discussions of the SD and GR approaches, respectively.

When writing applications or tools to manipulate both images and two-dimensional SDs, this crucial difference between the interfaces must be taken into account. While the underlying data is stored in row-major order in both cases, the API parameters are not expressed in the same way. Consider the example of an SD data set and GR image that are stored as identically-shaped arrays of X columns by Y rows and accessed via the SDreaddata and GRreadimage functions, respectively. Both functions take the parameters start, stride, and edge:

- For SDreaddata, those parameters are expressed in (y,x) or [row,column] order. For example, start[0] is the starting point in the Y dimension and start[1] is the starting point in the X dimension. The same ordering holds true for all SD data set manipulation functions.
- For GRreadimage, those parameters are expressed in (x,y) or [column,row] order. For example, start[0] is the starting point in the X dimension and start[1] is the starting point in the Y dimension. The same ordering holds true for all GR functions manipulating image data.

FORTRAN

integer function mgrdimg(ri_id, start, stride, edge, data)

integer ri_id, start(2), stride(2), edge(2)
<valid numeric data type> data(*)

integer function mgrcimg(ri_id, start, stride, edge, data)

integer ri_id, start(2), stride(2), edge(2)
character*(*) data
GRreadlut/mgrdlut/mgrclut

intn GRreadlut(int32 pal_id, VOIDP pal_data)

| pal_id   | IN:   | Palette identifier returned by GRgetlutid |
|pal_data  | OUT:  | Buffer for the palette data              |

Purpose Reads a palette.

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

Description GRreadlut reads the palette specified by the parameter pal_id into the buffer pal_data.

Note that there are two FORTRAN-77 versions of this routine; one for numeric data (mgrdlut) and the other for character data (mgrclut).

FORTRAN

```
integer function mgrdlut(pal_id, pal_data)
integer pal_id
<valid numeric data type> pal_data(*)

integer function mgrclut(pal_id, pal_data)
integer pal_id
character*(*) pal_data
```
GRreftoindex/mgr2idx

int32 GRreftoindex(int32 gr_id, uint16 gr_ref)

- **gr_id** IN: GR interface identifier returned by GRstart
- **gr_ref** IN: Reference number of the raster image

**Purpose**
Maps the reference number of a raster image to an index.

**Return value**
Returns the index of the image if successful and FAIL (or -1) otherwise.

**Description**
GRreftoindex returns the index of the raster image specified by the parameter gr_ref.

**FORTRAN**
integer function mgr2idx(gr_id, gr_ref)

integer gr_id, gr_ref
GRreqimageil/mgrimil

GRreqimageil(int32 ri_id, intn interlace_mode)

ri_id IN: Raster image identifier returned by GRcreate or GRselect

interlace_mode IN: Interlace mode

Purpose
Specifies the interlace mode to be used in the subsequent raster image read operation(s).

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

Description
GRreqimageil requests that the subsequent read operations on the image identified by the parameter ri_id use the interlace mode specified by the parameter interlace_mode.

The parameter interlace_mode specifies the interlace mode in which the data will be stored in memory when being read. Valid values of the parameter interlace_mode are MFGR_INTERLACE_PIXEL (or 0), MFGR_INTERLACE_LINE (or 1) and MFGR_INTERLACE_COMPONENT (or 2).

In the file, the image is always stored in pixel interlace mode, i.e. MFGR_INTERLACE_PIXEL. The interlace mode of the raster image specified at creation time is stored in the file along with the raster image. If GRreqimageil is not called prior to the call to GRreadimage, the raster image will be read and stored in memory according to the interlace mode specified at creation. If GRreqimageil is called before GRreadimage, GRreadimage will read the raster image and store it according to the interlace mode specified in the call to GRreqimageil.

FORTRAN
integer function mgrimil(ri_id, interlace_mode)

integer ri_id, interlace_mode
intn GRreqlutil(int32 ri_id, intn interlace_mode)

**ri_id**
IN: Raster image identifier returned by **GRcreate** or **GRselect**

**interlace_mode**
IN: Interlace mode

**Purpose**
Specifies the interlace mode to be used in the next palette read operation(s).

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

**Description**
**GRreqlutil** requests that the subsequent read operations on the palette attached to the image identified by the parameter **ri_id**, use the interlace mode **interlace_mode**.

The parameter **interlace_mode** specifies the interlace mode in which the data will be stored in memory when being read. Valid values of the parameter **interlace_mode** are **MFGR_INTERLACE_PIXEL** (or 0), **MFGR_INTERLACE_LINE** (or 1) and **MFGR_INTERLACE_COMPONENT** (or 2).

**FORTRAN**

integer function mglutil(ri_id, interlace_mode)

integer ri_id, interlace_mode
GRselect/mgselect

int32 GRselect(int32 gr_id, int32 index)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gr_id</td>
<td>IN: GR interface identifier returned by GRstart</td>
</tr>
<tr>
<td>index</td>
<td>IN: Index of the raster image in the file</td>
</tr>
</tbody>
</table>

**Purpose**
Selects the existing raster image.

**Return value**
Returns the raster image identifier if successful or FAIL (or -1) otherwise.

**Description**
GRselect obtains the identifier of the raster image specified by its index, index.

Valid values of the parameter index range from 0 to the total number of raster images in the file - 1. The total number of the raster images in the file can be obtained by using GRfileinfo.

**FORTRAN**
integer function mgselect(gr_id, index)

integer gr_id, index
GRsetattr/mgsnatt/mgscatt

intn GRsetattr(int32 [obj]_id, char *attr_name, int32 data_type, int32 count, VOIDP values)

{obj}_id IN: Raster image identifier (ri_id), returned by GRcreate or GRselect or
GR interface identifier (gr_id), returned by GRstart
attr_name IN: Name of the attribute
data_type IN: Data type of the attribute
count IN: Number of values in the attribute
values IN: Buffer for the attribute values

Purpose Assigns an attribute to a raster image or a file.
Return value Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.

Description GRsetattr attaches the attribute to the object specified by the parameter
objc_id. The attribute is defined by its name, attr_name, data type, data_type, number
of attribute values, count, and the attribute values, values. GRsetattr provides a generic
way for users to define metadata. It implements the label = value data abstraction.

If an GR interface identifier (gr_id) is specified as the parameter obj_id, a
global attribute is created which 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.

The parameter attr_name can be any ASCII string.

The parameter data_type can contain any data type supported by the HDF
library. These data types are listed in Table 1A in Section I of this manual.

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 FILL_ATTR definition.

Note that there are two FORTRAN-77 versions of this routine; one for numeric
data (mgsnatt) and the other for character data (mgscatt).

FORTRAN

integer function mgsnatt([obj]_id, attr_name, data_type, count,
values)

integer ri_id, comp_type, comp_prm(*)
integer [obj]_id, data_type, count
count(*) attr_name
<valid numeric data type> values(*)
integer function mgscatt([obj]_id, attr_name, data_type, count, values)

integer [obj]_id, data_type
integer count
character(*)(*) values, attr_name
GRsetcompress/mgscompress

```c
intn GRsetcompress(int32 ri_id, int32 comp_type, comp_info *c_info)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ri_id</code></td>
<td>Raster image identifier returned by GRcreate or GRselect</td>
</tr>
<tr>
<td><code>comp_type</code></td>
<td>Compression method for the image data</td>
</tr>
<tr>
<td><strong>C only:</strong></td>
<td>Pointer to the <code>comp_info</code> union</td>
</tr>
<tr>
<td><code>c_info</code></td>
<td>Compression parameters array</td>
</tr>
<tr>
<td><strong>Fortran only:</strong></td>
<td>Compression parameters array</td>
</tr>
</tbody>
</table>

**Purpose**

Specifies if the raster image will be stored in a file as a compressed raster image.

**Return value**

Returns `SUCCEED` (or 0) if successful and `FAIL` (or -1) otherwise.

**Description**

GRsetcompress specifies if the raster image specified by the parameter `ri_id` will be stored in the file in compressed format.

The compression method is specified by the parameter `comp_type`. Valid values of the parameter `comp_type` are:

- `COMP_CODE_NONE` (or 0) for no compression
- `COMP_CODE_RLE` (or 1) for RLE run-length encoding
- `COMP_CODE_SKPHUFF` (or 3) for Skipping Huffman compression
- `COMP_CODE_DEFLATE` (or 4) for GZIP compression
- `COMP_CODE_SZIP` (or 5) for Szip compression
- `COMP_CODE_JPEG` (or 7) for JPEG compression

The compression method parameters are specified by the parameter `c_info` in C and the parameter `comp prm` in Fortran. The parameter `c_info` has type `comp_info`, which is described in the `hcomp.h` header file. It contains algorithm-specific information for the library compression routines.

The skipping size for the Skipping Huffman algorithm is specified in the field `c_info.skphuff.skp_size` in C and in the parameter `comp prm(1)` in Fortran.

The deflate level for the GZIP algorithm is specified in the field `c_info.deflate.level` in C and in the parameter `comp prm(1)` in Fortran.
The parameter \( c_{\text{info}} \) is a pointer to a union structure of type \( \text{comp\_info} \). This union structure is defined as follows:

```c
typedef union tag_comp_info
{
  struct
  {
    /* Not used by GRsetcompress */
    } jpeg;

  struct
  {
    /* Not used by GRsetcompress */
    } nbit;

  struct
  { /* struct to contain info about how to compress size of the elements when skipping */
    int nskp_size;
  } skphuff;

  struct
  { /* struct to contain info about how to compress or decompress gzip encoded dataset how hard to work when compressing data*/
    int nlevel;
  } deflate;

  struct
  { /* struct to contain info about how to compress or decompress szip encoded dataset*/
    int32 bits_per_pixel;
    int32 compression_mode;
    int32 options_mask;
    int32 pixels;
    int32 pixels_per_block;
    int32 pixels_per_scanline;
  } szip;
}

} comp_info;
```

**FORTRAN**

```fortran
integer mgscompress(ri_id, comp_type, compprm)

integer ri_id, comp_type, compprm(*)
```
GRsetchunk/mgschnk

intn GRsetchunk(int32 ri_id, HDF_CHUNK_DEF cdef, int32 flags)

    ri_id          IN:     Raster image identifier returned by GRcreate or GRselect

    C only:

    cdef          IN:     Chunk definition

    flags         IN:     Compression flags

    Fortran only:

    dim_length    IN:     Chunk dimensions array

    comp_type     IN:     Type of compression

    comp_prm      IN:     Compression parameters array

Purpose

Makes a raster image a chunked raster image.

Return value

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

Description

GRsetchunk makes the raster image specified by the parameter ri_id a chunked raster image according to the chunking and compression information provided in the parameters cdef and flags in C, or in the parameters comp_type and comp_prm in Fortran.

C only:

The parameter cdef is a union of type HDF CHUNK DEF, which is defined as follows:

typedef union hdf_chunk_def_u
{
    int32 chunk_lengths[2]; /* chunk lengths along each dim */

    struct
    {
        int32 chunk_lengths[2];
        int32 comp_type;      /* compression type */
        struct comp_info cinfo;
    } comp;

    struct
    { /* is not used in GR interface */
        nbit;
    } nbit;
} HDF CHUNK DEF
Purpose Makes a raster image a chunked raster image.

Valid values of the parameter flags are HDF_CHUNK for chunked and uncompressed data and (HDF_CHUNK | HDF_COMP) for chunked and compressed data. Data can be compressed using run-length encoding (RLE), Skipping Huffman or GZIP compression algorithms.

If the parameter flags has a value of HDF_CHUNK, the chunk dimensions must be specified in the field cdef.chunk_lengths[]. If the parameter flags has a value of (HDF_CHUNK | HDF_COMP), the following must be specified:

1) The chunk dimensions in the field cdef.comp.chunk_lengths[].
2) The compression type in the field cdef.comp.comp_type. Valid values of compression type values are listed below.

COMP_CODE_NONE (or 0) for uncompressed data
COMP_CODE_RLE (or 1) for data compressed using the RLE compression algorithm
COMP_CODE_SKPHUFF (or 3) for data compressed using the Skipping Huffman compression algorithm
COMP_CODE_DEFLATE (or 4) for data compressed using the GZIP compression algorithm
COMP_CODE_SZIP (or 5) for data compressed using the Szip compression algorithm

3) If using Skipping Huffman compression, the skipping size is specified in the field cdef.comp.cinfo.skphuff.skp_size. If using GZIP compression, the deflate level is specified in the field cdef.comp.cinfo.deflate.level. Valid deflate level values are integers from 1 to 9 inclusive.

Refer to the SDsetcompress page in this manual for the definition of the comp_info structure and for setting Szip compression parameters.

Fortran only:

The dim_length array specifies the chunk dimensions.

The parameter comp_type specifies the compression type. Valid compression types and their values used are defined in the hdf.inc file, and are listed below.
• COMP_CODE_NONE (or 0) for uncompressed data
• COMP_CODE_RLE (or 1) for data compressed using the RLE compression algorithm
• COMP_CODE_SKPHUFF (or 3) for data compressed using the Skipping Huffman compression algorithm
• COMP_CODE_DEFLATE (or 4) for data compressed using the GZIP compression algorithm.
• COMP_CODE_SZIP (or 5) for data compressed using the Szip compression algorithm.

The parameter comp_prm 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.

See the SDsetcompress reference manual entry for a description of the struct required with the use of Szip compression.

FORTRAN

integer function mgschnk(ri_id, dim_length, comp_type, comp_prm)

integer ri_id, dim_length, comp_type, comp_prm
GRsetchunkcache/mgscchnk

intn GRsetchunkcache(int32 ri_id, int32 maxcache, int32 flags)

ri_id IN: Raster image identifier returned by GRcreate or GRselect
maxcache IN: Maximum number of chunks to cache
flags IN: Flags determining the behavior of the routine

Purpose Specifies the maximum number of chunks to cache.

Return value Returns the value of the parameter maxcache if successful and FAIL (or -1) otherwise.

Description GRsetchunkcache sets the maximum number of chunks to be cached for the chunked raster image specified by the parameter ri_id. The maximum number of the chunks is specified by the parameter maxcache.

Currently, the only valid value of the parameter flags is 0.

If GRsetchunkcache is not called, the maximum number of chunks in the cache is set to the number of chunks along the fastest-changing dimension. Refer to the discussion of the GRsetchunkcache routine in the HDF User's Guide for more specific information on the routine’s behavior.

FORTRAN integer function mgscchnk(ri_id, maxcache, flags)

integer ri_id, maxcache, flags
GRsetexternalfile/mgsxfil

intn GRsetexternalfile(int32 ri_id, char *filename, int32 offset)

   ri_id    IN: Raster image identifier returned by GRcreate or GRselect
   filename IN: Name of the external file
   offset   IN: Offset in bytes from the beginning of the external file to where the
data will be written

Purpose Specifies that the raster image will be written to an external file.
Return value Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description GRsetexternalfile specifies that the raster image identified by the parameter
ri_id will be written to the external file specified by the parameter filename at
the offset specified by the parameter offset.

Data can only be moved 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. To release
the space in the primary file use the hdfpack command-line utility. If the
raster image does not exist, its data will be written to the external file during
the subsequent calls to GRwritedata.

See the reference manual entries for HXsetcreatedir and HXsetdir for more
information on the options available for accessing external files.

FORTRAN  
integer function mgsxfil(ri_id, filename, offset)

integer ri_id, offset
character*(*) filename
**GRstart/mgstart**

```c
int32 GRstart(int32 file_id)
```

- **file_id**  
  IN: File identifier returned by `Hopen`

**Purpose**  
Initializes the GR interface.

**Return value**  
Returns the GR interface identifier if successful and `FAIL` (or -1) otherwise.

**Description**  
`GRstart` initializes the GR interface for the file specified by the parameter `file_id`.

This routine is used with the `GRend` routine to define the extent of the GR interface session. As with the start routines in the other interfaces, `GRstart` initializes the internal interface structures needed for the remaining GR routines. Use the general purpose routines `Hopen` and `Hclose` to manage file access. The GR routines will not open and close HDF files.

**FORTRAN**

```fortran
integer function mgstart(file_id)

integer file_id
```
GRwritechunk/mgwchnk/mgwcchnk

intn GRwritechunk(int32 ri_id, int32 *origin, const VOIDP datap)

ri_id IN: Raster image identifier returned by GRcreate or GRselect
origin IN: Origin of the chunk to be written
datatap IN: Buffer for the chunk to be written

Purpose Writes a data chunk to a chunked raster image (pixel-interlace only)

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

GRwritechunk returns FAIL (or -1) when an attempt is made to use it to write to a non-chunked raster image.

Description GRwritechunk writes the entire chunk of data stored in the buffer datap to the chunked raster image identified by the parameter ri_id. Writing starts at the location specified by the parameter origin. This function has less overhead than GRwriteimage and should be used whenever an entire chunk of data is to be written.

The parameter origin is a two-dimensional array which specifies the coordinates of the chunk according to the chunk position in the overall chunk array. Refer to Chapter 8, "General Raster Images (GR API)," in the HDF User’s Guide.

The datap buffer contains the chunk’s data organized in a pixel interlace mode.

FORTRAN

integer mgwchnk(ri_id, origin, datap)
integer ri_id, origin(2)
<valid_numeric_datatype> datap(*)

integer mgwcchnk(ri_id, origin, char_datap)
integer ri_id, origin(2)
character(*) char_datap
GRwriteimage/mgwrimg/mgwciimg

intn GRwriteimage(int32 ri_id, int32 start[2], int32 stride[2], int32 edge[2], VOIDP data)

ri_id IN: Raster image identifier returned by GRcreate or GRselect
start IN: Array containing the two-dimensional coordinate of the initial location for the write
stride IN: Array containing the number of data locations the current location is to be moved forward before each write
edge IN: Array containing the number of data elements that will be written along each dimension
data IN: Buffer containing the image data

Purpose Writes a raster image.

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

Description GRwriteimage writes the subsample of the raster image data stored in the buffer data to the raster image specified by the parameter ri_id. 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. The first element of the array start specifies an offset from the beginning of the array data along the fastest-changing dimension, which is the second dimension in C and the first dimension in Fortran. The second element of the array start specifies an offset from the beginning of the array data along the second fastest-changing dimension, which is the first dimension in C and the second dimension in Fortran. For example, if the first value of the array start is 2 and the second value is 3, the starting location of the subsample to be written is at the fourth row and third column in C, and at the third row and fourth column in Fortran.

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 edge specifies the number of data elements to be written along the corresponding dimension. The correspondence between the elements of the array edge and the dimensions of the array data is the same as described above for the array start.

Note that there are two FORTRAN-77 versions of this routine; one for numeric data (mgwrimg) and the other for character data (mgwcimg).
Note

Regarding an important difference between the SD and GR interfaces:
The SD and GR interfaces differ in the correspondence between the dimension order in parameter arrays such as start, stride, edge, and dimsizes and the dimension order in the data array. See the SDreaddata and GRreadimage reference manual pages for discussions of the SD and GR approaches, respectively.

When writing applications or tools to manipulate both images and two-dimensional SDs, this crucial difference between the interfaces must be taken into account. While the underlying data is stored in row-major order in both cases, the API parameters are not expressed in the same way. Consider the example of an SD data set and GR image that are stored as identically-shaped arrays of X columns by Y rows and accessed via the SDreaddata and GRreadimage functions, respectively. Both functions take the parameters start, stride, and edge:

- For SDreaddata, those parameters are expressed in (y,x) or [row,column] order. For example, start[0] is the starting point in the Y dimension and start[1] is the starting point in the X dimension. The same ordering holds true for all SD data set manipulation functions.
- For GRreadimage, those parameters are expressed in (x,y) or [column,row] order. For example, start[0] is the starting point in the X dimension and start[1] is the starting point in the Y dimension. The same ordering holds true for all GR functions manipulating image data.

FORTRAN

integer function mgwrimg(ri_id, start, stride, edge, data)

integer ri_id, start(2), stride(2), edge(2)
<valid numeric data type> data(*)

integer function mgwcimg(ri_id, start, stride, edge, data)

integer ri_id, start(2), stride(2), edge(2)
character*(*) data
GRwritelut/mgwrlut/mgwclut

intn GRwritelut(int32 pal_id, int32 ncomp, int32 data_type, int32 interlace_mode, int32 num_entries, VOIDP pal_data)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pal_id</td>
<td>IN: Palette identifier returned by GRgetlutid</td>
</tr>
<tr>
<td>ncomp</td>
<td>IN: Number of components in the palette</td>
</tr>
<tr>
<td>data_type</td>
<td>IN: Data type of the palette data</td>
</tr>
<tr>
<td>interlace_mode</td>
<td>IN: Interlace mode of the stored palette data</td>
</tr>
<tr>
<td>num_entries</td>
<td>IN: Number of entries in the palette</td>
</tr>
<tr>
<td>pal_data</td>
<td>IN: Buffer for the palette data to be written</td>
</tr>
</tbody>
</table>

Purpose

Writes a palette.

Return value

Returns succeed (or 0) if successful and fail (or -1) otherwise.

Description

GRwritelut writes a palette with the number of pixel components specified by the parameter ncomp, the data type of the palette data specified by the parameter data_type, the interlace mode specified by the parameter interlace_mode, and the number of entries in the palette specified by the parameter num_entries. The palette data itself is stored in the pal_data buffer. Currently only “old-style” palettes are supported, i.e. ncomp = 3, num_entries = 256, data_type = uint8.

The parameter ncomp specifies the number of pixel components in the palette and must have a value of at least 1.

The parameter data_type specifies the type of the palette data and can be any of the data types supported by the HDF library. The data types supported by HDF are listed in Table 1A in Section I of this manual.

The parameter interlace_mode specifies the interlacing in which the palette is to be written. The valid values of interlace_mode are: MFGR_INTERLACE_PIXEL (or 0), MFGR_INTERLACE_LINE (or 1) and MFGR_INTERLACE_COMPONENT (or 2).

The buffer pal_data is assumed to have sufficient space allocated to store all of the palette data.

Note that there are two FORTRAN-77 versions of this routine; one for numeric data (mgwrlut) and the other for character data (mgwclut).

FORTRAN

integer function mgwrlut(pal_id, ncomp, data_type, interlace_mode, num_entries, pal_data)

integer pal_id, ncomp, data_type, interlace_mode, num_entries

<valid numeric data type> pal_data(*)
integer function mgwclut(pal_id, ncomp, data_type, interlace_mode, num_entries, pal_data)

integer pal_id, ncomp, data_type, interlace_mode, num_entries

character(*) pal_data
Hclose/hclose

```c
int Hclose(int32 file_id)
```

**Purpose**
Closes the access path to the file.

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

**Description**
The file identifier `file_id` is validated before the file is closed. If the identifier is valid, the function closes the access path to the file.

If there are still access identifiers attached to the file, the error `DFE_OPENAID` is placed on the error stack, `FAIL` (or -1) is returned, and the file remains open. This is a common error when developing new interfaces. Refer to the Reference Manual page on `HendAccess` for a discussion of this problem.

```fortran
integer function hclose(file_id)
  integer file_id
```

December 30, 2003

2-73
Hgetfileversion/hgfilever

intn Hgetfileversion(int32 file_id, uint32 *major_v, uint32 *minor_v, uint32 *release, char string[])

file_id IN: File identifier returned by Hopen
major_v OUT: Major version number
minor_v OUT: Minor version number
release OUT: Release number
string OUT: Version number text string

Purpose
Retrieves version information for an HDF file.

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

Description
It is still an open question as to what exactly the version number of a file should mean, so we recommend that code not depend on this buffer. The string argument is limited to a length of LIBVSTR_LEN (or 80) characters as defined in hfile.h.

FORTRAN
integer function hgfilever(file_id, major_v, minor_v, release, string)

integer file_id, major_v, minor_v, release
character*(*) string
**Hgetlibversion/hglibver**

```c
intn Hgetlibversion(uint32 *major_v, uint32 *minor_v, uint32 *release, char string[])
```

- `major_v` OUT: Major version number
- `minor_v` OUT: Minor version number
- `release` OUT: Release number
- `string` OUT: Version number text string

**Purpose**
Retrieves the version information of the current HDF library.

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

**Description**
The version information is compiled into the HDF library, so it is not necessary to have any open files for this function to execute. The `string` buffer is limited to a length of LIBVSTR_LEN (or 80) characters as defined in `hfile.h`.

**FORTRAN**
```fortran
integer function hglibver(major_v, minor_v, release, string)

integer major_v, minor_v, release
character(*) string
```
Hishdf/hishdff

Hishdf/hishdff

intn Hishdf(char *filename)

filename IN: Complete path and filename of the file to be checked

**Purpose** Determines if a file is an HDF file.

**Return value** Returns **TRUE** (or 1) if the file is an HDF file and **FALSE** (or 0) otherwise.

**Description** The first four bytes of a file identify it as an HDF file. It is possible that **Hishdf** will identify a file as an HDF file but **Hopen** will be unable to open the file; for example, if the data descriptor list is corrupt.

**Fortran**

```fortran
integer function hishdff(filename)
character*(*) filename
```
Hopen/hopen

```c
int32 Hopen(char *filename, intn access, int16 n_dds)
```

**filename**

IN: Complete path and filename for the file to be opened

**access**

IN: Access code definition (preceded by DFACC_)

**n_dds**

IN: Number of data descriptors in a block if a new file is to be created

**Purpose**

Provides an access path to an HDF file by reading all the data descriptor blocks into memory.

**Return value**

Returns the file identifier if successful and FAIL (or -1) otherwise.

**Description**

If given a new file name, **Hopen** will create a new file using the specified access type and number of data descriptors. If given an existing file name, **Hopen** will open the file using the specified access type and ignore the n_dds argument.

The number of data descriptors in a block, n_dds, is a non-negative integer with a default value of DEF_NDDS (or 16) and a minimum value of MIN_NDDS (or 4). If the specified value of n_dds is less than MIN_NDDS, then it will be set to MIN_NDDS.

HDF provides several access code definitions:

- **DFACC_CREATE**: If file exists, delete it, then open a new file for read/write.
- **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.

If a file is opened and an attempt is made to reopen the file using DFACC_CREATE, HDF will issue the error code DFE_ALROPEN. If the file is opened with read-only access and an attempt is made to reopen the file for write access using DFACC_WRITE, HDF will attempt to reopen the file with read and write permissions.

Upon successful exit, the specified file is opened with the relevant permissions, the data descriptors are set up in memory, and the associated file_id is returned. For new files, the appropriate file headers are also set up.

**FORTRAN**

```fortran
integer function hopen(filename, access, n_dds)
character*(*) filename
integer access, n_dds
```

HCgetcompress

intn HCgetcompress(int32 access_id, comp_coder_t *comp_type, comp_info *c_info)

access_id        IN:  Access record identifier
comp_type        OUT:  Type of compression
c_info           OUT:  Pointer to compression information structure

Purpose          Retrieves data compression type and information for specified element.
Return value     Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description      HCgetcompress retrieves the compression type and compression information for the element specified by access_id.

The compression method is returned in the parameter comp_type. Valid values of comp_type are as follows:

- COMP_CODE_NONE (or 0) for no compression
- COMP_CODE_RLE (or 1) for RLE run-length encoding
- COMP_CODE_NBIT (or 2) for NBIT compression
- COMP_CODE_SKPHUFF (or 3) for Skipping Huffman compression
- COMP_CODE_DEFLATE (or 4) for GZIP compression
- COMP_CODE_JPEG (or 6) for JPEG compression

The compression method parameters are returned in the c_info struct in C and the array parameter comp prm in Fortran. The c_info struct is of type comp_info, contains algorithm-specific information for the library compression routines, and is described in the hcomp.h header file.

HCgetcompress is used by GRgetcompress and SDgetcompress.

Note regarding HDF4 Release 2.0 beta: Szip compression has been added to the HDF4 library for Release 2.0. That addition is documented in the GRsetcompress and GRgetcompress reference manual entries but is not yet reflected here.
## HDdont_atexit/hddontatexit

The function `HDdont_atexit` takes no arguments and returns an integer value.

### Purpose
Indicates to the library that an `atexit()` routine is _not_ to be installed.

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

### Description
This routine indicates to the library that an `atexit()` cleanup routine should not be installed. The purpose for this is in situations where the library is dynamically linked into an application and is unlinked from the application before `exit()` gets called. In those situations, a routine installed with `atexit()` would jump to a routine which was no longer in memory, causing errors.

In order to be effective, this routine _must_ be called before any other HDF function calls, and _must_ be called each time the library is loaded/linked into the application (the first time and after it has been unloaded).

If this routine is used, certain memory buffers will not be deallocated, although in theory a user could call `HPend` on their own.

### FORTRAN

```fortran
integer hddontatexit()
```
**HEprint/heprntf/heprnt**

VOID HEprint(FILE *\textit{stream}, int32 \textit{level})

\begin{itemize}
  \item \textit{stream} \ \textbf{IN}: Stream to print error message to
  \item \textit{level} \ \textbf{IN}: Level of error stack to print
\end{itemize}

**Purpose**
Prints information to the error stack.

**Return value**
None.

Fortran function returns 0 (zero) on success or -1 on failure.

**Description**
If \textit{level} is 0, all of the errors currently on the error stack are printed. Output from this function is sent to the file pointed to by \textit{stream}.

The following information is printed: the ASCII description of the error, the reporting routine, the reporting routine as source file name, and the line at which the error was reported. If the programmer has supplied extra information by means of \textbf{HEreport}, this information is printed as well.

The FORTRAN-77 routine \textbf{heprnt} uses one less parameter than the C routine because it doesn't allow the user to specify the print stream. Instead, it always prints to stdout.

The FORTRAN-77 routine \textbf{heprntf} is available on all platforms; \textbf{heprnt} is not supported on Microsoft Windows platforms.

The \textbf{heprntf} parameter \textit{filename} is the name of the file to which error output is to be written. If the value of \textit{filename} is an empty string (''), error output will be written to standard output, stdout.

**FORTRAN**

\begin{verbatim}
integer function heprntf(filename, level)
character(*) filename
integer level

integer function heprnt(level)
integer level
\end{verbatim}
HEstring/hestringf

const char *HEstring(hdf_err_code_t error_code)

error_code IN: HDF error code

Purpose  Returns the error message associated with specified error code.

Return value  Returns a pointer to a string associated with the error code if successful.

Description  Returns a text description of the given error code. These strings are statically declared and should not be deallocated from memory (using the free routine) by the user. If a defined text description cannot be found a generic default message is returned.

FORTRAN  integer function hestringf(error_code, error_message)

integer error_code
character(*) error_message
HXsetcreatedir/hxiscdir

intn HXsetcreatedir(char *dir)

*dir* IN: Target directory of the external file to be written

**Purpose**
Initializes the directory environment variable, identifying the location of the external file to be written.

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

**Description**
The contents of *dir* is copied into the private memory of the HDF library. If *dir* is NULL, the directory variable is unset. If HXsetcreatedir encounters an error condition, the directory variable is not changed. When a new external element is created (via the routines HXcreate or SDsetexternal), the HDF library accesses the external file just like the open call by default. Refer to the Reference Manual page on HXcreate for a description of when a new or an old file should be opened.

Users may override the default action by calling HXsetcreatedir or by defining the environment variable $HDFEXTCREATEDIR. The HDF library will access the external file in the directory according to the environment variable setting. The precedence is HXsetcreatedir, then $HDFEXTDIR, in the manner of open.

Note that the above override does not apply to absolute pathnames - i.e., filenames starting with a forward slash. HDF will access the absolute pathname without change. Also note that HXsetcreatedir and $HDFEXTCREATEDIR are not symmetrical to HXsetdir and $HDFEXTDIR. The former pair permits only single directory values and is used to compose the filename for access. The later pair permits multiple directory values which are used for searching an existing file.

The *dir_len* parameter in the FORTRAN-77 routine specifies the length of the *dir* character string.

**FORTRAN**

integer function hxiscdir(dir, dir_len)

character*(*) dir

integer dir_len
**HXsetdir/hxisdir**

intn HXsetdir(char *dir)

*dir* IN: Target directory of the external file to be located

**Purpose**
Initializes the directory environment variable, identifying the location of the external file to be located.

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

**Description**
HXsetdir sets the directory variable for locating an external file according to *dir* which may contain multiple directories separated by vertical bars (e.g., “dir1|dir2”). The content of *dir* is copied into the private memory of the HDF library. If *dir* is NULL, the directory variable is unset.

If HXsetdir encounters any error, the directory variable is not changed. By default, the HDF library locates the external file just like the open call. It also searches for the external file in the directories specified by the user environment variable $HDFEXTDIR, if defined, and the directory variable set by HXsetdir. The searching precedence is directory variable, if set, then $HDFEXTDIR, then in the manner of open.

The searching differs if the external filename is an absolute pathname - i.e., starting with a forward slash. HDF will try open first. If open fails and if $HDFEXTDIR is defined or the directory variable is set via HXsetdir, HDF will remove all directory components of the absolute pathname (e.g., “/usr/groupA/projectB/Data001” becomes “Data001”) and search for that filename with the strategy described in the previous paragraph.

The *dir_len* parameter in the FORTRAN-77 routine specifies the length of the *dir* character string.

**FORTRAN**

```fortran
integer function hxisdir(dir, dir_len)
character*(*) dir
integer dir_len
```

December 30, 2003
**SDattrinfo/sfgainfo**

```c
intn SDattrinfo(int32 obj_id, int32 attr_index, char *attr_name, int32 *data_type, int32 *count)
```

- **obj_id**  
  IN: Identifier of the object to which the attribute is attached to

- **attr_index**  
  IN: Index of the attribute

- **attr_name**  
  OUT: Name of the attribute

- **data_type**  
  OUT: Data type of the attribute values

- **count**  
  OUT: Total number of values in the attribute

**Purpose**  
Retrieves information about an attribute.

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

**Description**  
SDattrinfo retrieves the name, data type, and number of values of the attribute specified by its index, `attr_index`, and stores them in the parameters `attr_name`, `data_type`, and `count`, respectively. This routine should be used before reading the values of an attribute with `SDreadattr`.

The parameter `obj_id` can be either an SD interface identifier (`sd_id`), returned by `SDstart`, a data set identifier (`sds_id`), returned by `SDselect`, or a dimension identifier (`dim_id`), returned by `SDgetdimid`.

Valid values of the parameter `attr_index` range from 0 to the number of attributes attached to the object - 1.

Valid values of the parameter `data_type` can be found in Table 1A of Section I of this manual.

**FORTRAN**

```fortran
integer function sfgainfo(obj_id, attr_index, attr_name, data_type, count)
character*(*) attr_name
integer obj_id, attr_index, data_type, count
```
### SDcheckempty/sfcheckempty

**int32 SDcheckempty( int32 sds_id, intn *emptySDS )**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sds_id</td>
<td>IN: SDS identifier</td>
</tr>
<tr>
<td>emptySDS</td>
<td>OUT: Boolean value indicating whether the SDS is empty</td>
</tr>
</tbody>
</table>

**Purpose**
Determines whether a scientific dataset (an SDS) is empty.

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

**Description**
SDcheckempty sets the parameter emptySDS to TRUE if the dataset identified by sds_id has not been written with data, and to FALSE, otherwise.

The Fortran routine, sfcheckempty, returns 1 in emptySDS if the dataset is empty and 0 otherwise.

**FORTRAN**

```fortran
integer function sfcheckempty(sds_id, emptySDS)
integer sds_id, emptySDS
```
**SDcreate/sfcreate**

int32 SDcreate(int32 sd_id, char *name, int32 data_type, int32 rank, int32 dimsizes[])

- **sd_id** IN: SD interface identifier returned by **SDstart**
- **name** IN: Name of the data set
- **data_type** IN: Data type for the values in the data set
- **rank** IN: Number of the data set dimensions
- **dimsizes** IN: Array containing the size of each dimension

**Purpose**
Creates a new data set.

**Return value**
Returns the data set identifier (sds_id) if successful and FAIL (or -1) otherwise.

**Description**
**SDcreate** creates a data set with the name specified by the parameter name, the values of the data type specified by parameter data_type, the number of dimensions specified by the parameter rank, and the dimension sizes specified by the array dimsizes.

Once a data set has been created, it is not possible to change its name, data type, or rank. However, it is possible to create a data set and close the file before writing any data values to it. The values can be added or modified at a future time. To add data or modify an existing data set, use **SDselect** to get the data set identifier instead of **SDcreate**.

If the parameter name is NULL in C or an empty string in Fortran, the default name “Data Set” will be generated. The length of the name specified by the name parameter must be 63 characters or less.

- If the length of the name is longer than 63 characters but less than or equal to MAX_NC_NAME characters (256 characters by default), then the name will be truncated to 63 characters.
- If the length of the name is longer than MAX_NC_NAME characters (256 characters by default), then **SDcreate** will fail.

The calling program must ensure that the length of the dimsizes array is the value of the rank parameter, which is between 1 and MAX_VAR_DIMS (or 32).

To create a data set with an unlimited dimension, assign the value of SD_UNLIMITED (or 0) to dimsizes[0] in C and to dimsizes(rank) in Fortran.

The data_type parameter can contain any data type supported by the HDF library. These data types are listed in Table 1A in Section I of this manual.

See the notes regarding the potential performance impact of unlimited dimension data sets in the **HDF User’s Guide** Section 14.4.3, "Unlimited Dimension Data Sets (SDSs and Vdatas) and Performance."
Note

Regarding an important difference between the SD and GR interfaces:
The SD and GR interfaces differ in the correspondence between the dimension order
in parameter arrays such as `start`, `stride`, `edge`, and `dimsizes` and the dimension order
in the `data` array. See the `SDreaddata` and `GRreadimage`
reference manual pages for discussions of the SD and GR approaches,
respectively.

When writing applications or tools to manipulate both images and two-
dimensional SDs, this crucial difference between the interfaces must be taken
into account. While the underlying data is stored in row-major order in both
cases, the API parameters are not expressed in the same way. Consider the
example of an SD data set and GR image that are stored as identically-shaped
arrays of X columns by Y rows and accessed via the `SDreaddata` and
`GRreadimage` functions, respectively. Both functions take the parameters
`start`, `stride`, and `edge`.

- For `SDreaddata`, those parameters are expressed in (y,x) or
  `[row,column]` order. For example, `start[0]` is the starting point in the
  Y dimension and `start[1]` is the starting point in the X dimension.
  The same ordering holds true for all SD data set manipulation
  functions.
- For `GRreadimage`, those parameters are expressed in (x,y) or
  `[column,row]` order. For example, `start[0]` is the starting point in the
  X dimension and `start[1]` is the starting point in the Y dimension.
  The same ordering holds true for all GR functions manipulating image
  data.

FORTRAN

```fortran
integer function sfcreate(sd_id, name, data_type, rank,
        dimsizes)

    character(*) name

    integer sd_id, data_type, rank, dimsizes(*)
```
**SDdiminfo/sfgdinfo**

```c
intn SDdiminfo(int32 dim_id, char *name, int32 *size, int32 *data_type, int32 *num_attrs)
```

- **dim_id** | IN: Dimension identifier returned by SDgetdimid
- **name** | OUT: Name of the dimension
- **size** | OUT: Size of the dimension
- **data_type** | OUT: Data type of the dimension scale
- **num_attrs** | OUT: Number of attributes assigned to the dimension

**Purpose**
Retrieves information about a dimension.

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

**Description**
SDdiminfo retrieves the name, size, data type, and number of values of the dimension specified by the parameter `dim_id`, and stores them in the parameters `name`, `size`, `data_type`, and `num_attrs`, respectively.

If the output value of the parameter `size` is set to 0, then the dimension specified by the `dim_id` parameter is unlimited. To get the number of records of an unlimited dimension, use SDgetinfo.

If scale information has been stored for this dimension via SDsetdimscale, the `data_type` parameter will contain the data type. Valid data types can be found in Table 1A of Section I of this manual. If no scale information has been stored for this dimension, the value returned in the `data_type` parameter will be 0.

If the user has not named the dimension via SDsetdimname, a default dimension name of “fakeDim[x]” will be generated by the library, where [x] denotes the dimension index. If the name is not desired, the parameter `name` can be set to NULL in C and an empty string in Fortran.

**FORTRAN**
```fortran
integer function sfgdinfo(dim_id, name, size, data_type, numAttrs)

character(*) name

integer dim_id, size, data_type, numAttrs
```
**SDend/sfend**

intn SDend(int32 sd_id)

\[ sd_id \quad \text{IN:} \quad \text{SD interface identifier returned by SDstart} \]

**Purpose**
Terminates access to an SD interface.

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

**Description**
SDend closes the file and frees memory allocated by the library when SD interface activities are completed. If the calling program exits without invoking this routine, recent changes made to the in-core file data are likely not to be flushed to the file. Note that each SDstart must have a matching SDend.

**FORTRAN**

integer function sfend(sd_id)

integer sd_id
SDendaccess/sfendacc

intn SDendaccess(int32 sds_id)

\begin{description}
\item[\textbf{sds}_id] \quad \text{IN:} \quad \text{Data set identifier returned by \texttt{SDcreate} or \texttt{SDselect}}
\item[\textbf{Purpose}] \quad \text{Terminates access to a data set.}
\item[\textbf{Return value}] \quad \text{Returns \texttt{SUCCEED} (or 0) if successful and \texttt{FAIL} (or -1) otherwise.}
\item[\textbf{Description}] \quad \texttt{SDendaccess} frees the memory taken up by the HDF library’s data structures devoted to the data set identified by the parameter \texttt{sds}_id.

Failing to call this routine after all operations on the specified data set are complete may result in loss of data. This routine must be called once for each call to \texttt{SDcreate} or \texttt{SDselect}.
\end{description}

\textbf{FORTRAN} \quad \texttt{integer function sfendacc(sds_id)}

\begin{verbatim}
integer sds_id
\end{verbatim}
SDfileinfo/sffinfo

intn SDfileinfo(int32 sd_id, int32 *num_datasets, int32 *num_global_attr)

sd_id IN: SD interface identifier returned by SDstart
num_datasets OUT: Number of data sets in the file
num_global_attr OUT: Number of global attributes in the file

Purpose
Retrieves the number of data sets and the number of global attributes in a file.

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

Description
SDfileinfo returns the number of data sets in the parameter num_datasets and
the number of global attributes in the parameter num_global_attr. The term
“global attributes” refers to attributes that are assigned to the file. The global
attributes are created by SDsetattr using an SD interface identifier (sd_id)
rather than a data set identifier (sds_id).

The value returned by the parameter num_datasets includes the number of
coordinate variable data sets. To determine if the data set is a coordinate
variable, use SDcoordvar.

FORTRAN
integer function sffinfo(sd_id, num_datasets, num_global_attr)

integer sd_id, num_datasets, num_global_attr
SDfindattr/sffattr

int32 SDfindattr(int32 obj_id, char *attr_name)

- **obj_id** (IN): Identifier of the object to which the attribute is attached
- **attr_name** (IN): Name of the attribute

**Purpose**: Finds the index of an attribute given its name.

**Return value**: Returns the index if successful and **FAIL** (or -1) otherwise.

**Description**: 
SDfindattr retrieves the index of the object’s attribute with the name specified by the parameter **attr_name**.

The attribute is attached to the object specified by the parameter **obj_id**. The parameter **obj_id** can be either an SD interface identifier (**sd_id**), returned by **SDstart**, a data set identifier (**sds_id**), returned by **SDselect**, or a dimension identifier (**dim_id**), returned by **SDgetdimid**.

Wildcard characters are not allowed in the parameter **attr_name**. **SDfindattr** searches for the name specified in the parameter **attr_name** in a case-sensitive manner.

**FORTRAN**

```fortran
integer function sffattr(obj_id, attr_name)

integer obj_id
character(*) attr_name
```
SDgetcal/sfgcal

SDgetcal(int32 sds_id, float64 *cal, float64 *cal_err, float64 *offset, float64 *offset_err, int32 *data_type)

sds_id IN: Data set identifier returned by SDcreate or SDselect

cal OUT: Calibration factor

cal_err OUT: Calibration error

offset OUT: Uncalibrated offset

offset_err OUT: Uncalibrated offset error

data_type OUT: Data type of uncalibrated data

Purpose Retrieves the calibration information associated with a data set.

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

Description SDgetcal reads the calibration record attached to the data set identified by the parameter sds_id. A calibration record is comprised of four 64-bit floating point values followed by a 32-bit integer. The information is listed in the following table:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cal</td>
<td>Calibration factor</td>
</tr>
<tr>
<td>cal_err</td>
<td>Calibration error</td>
</tr>
<tr>
<td>offset</td>
<td>Uncalibrated offset</td>
</tr>
<tr>
<td>offset_err</td>
<td>Uncalibrated offset error</td>
</tr>
<tr>
<td>data_type</td>
<td>Data type of the uncalibrated data</td>
</tr>
</tbody>
</table>

The relationship between a calibrated value cal_value and the original value orig_value is defined as orig_value = cal * (cal_value - offset).

The variable offset_err contains a potential error of offset, and cal_err contains a potential error of cal. Currently the calibration record is provided for information only. The SD interface performs no operations on the data based on the calibration tag.

FORTRAN integer function sfgcal(sds_id, cal, cal_err, offset, offset_err, data_type)

integer sds_id, data_type

real*8 cal, cal_err, offset, offset_err
**SDgetchunkinfo/sfgichnk**

```c
intn SDgetchunkinfo(int32 sds_id, HDF_CHUNK_DEF *cdef, int32 *flag)
```

- **sds_id** IN: Data set identifier returned by SDcreate or SDselect

  **C only:**
  - **cdef** OUT: Pointer to the chunk definition
  - **flag** OUT: Compression flag

  **Fortran only:**
  - **dim_length** OUT: Array of chunk dimensions
  - **flag** OUT: Compression flag

**Purpose**
Retrieves chunking information for a data set.

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

**Description**
SDgetchunkinfo retrieves chunking information about the data set identified by the parameter sds_id into the parameters cdef and flag in C, and to the parameters dim_length and flag in Fortran.

Currently, only information about chunk dimensions is retrieved into the corresponding cdef structure element for each type of compression in C, and in the dim_length array in Fortran. No information on compression parameters is available in the comp structure of the HDF_CHUNK_DEF union. Refer to the page on SDsetchunk in this manual for specific information on the HDF_CHUNK_DEF union.

The value returned in the flag parameter indicates the data set type (i.e., if the data set is not chunked, chunked, and chunked and compressed).

If the chunk length for each dimension is not needed, NULL can be passed in as the value of the cdef parameter in C.

The following table shows the type of the data set, possible values of the flag parameter, and the corresponding cdef structure element filled with the chunk’s dimensions.

<table>
<thead>
<tr>
<th>Type of Data Set</th>
<th>Values of flag in C (Fortran)</th>
<th>cdef Structure Element Filled with the Chunk’s Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not chunked</td>
<td>HDF_NONE (-1)</td>
<td>None</td>
</tr>
<tr>
<td>Chunked</td>
<td>HDF_CHUNK (0)</td>
<td>cdef.chunk_lengths[]</td>
</tr>
<tr>
<td>Type of Data Set</td>
<td>Values of flag in C (Fortran)</td>
<td>cdef Structure Element Filled with the Chunk’s Dimensions</td>
</tr>
<tr>
<td>-----------------------------------------------------------</td>
<td>-------------------------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>Chunked and compressed with either the run-length encoding (RLE), Skipping Huffman or GZIP compression algorithms</td>
<td>HDF_CHUNK</td>
<td>HDF_COMP (1)</td>
</tr>
<tr>
<td>Chunked and compressed with NBIT compression</td>
<td>HDF_CHUNK</td>
<td>HDF_NBIT (2)</td>
</tr>
</tbody>
</table>

FORTRAN

```
integer function sfgichnk(sds_id, dim_length, flag)

integer sds_id, dim_length(*), flag
```
**SDgetcompress/sfgcompress**

```c
int SDgetcompress(int32 sds_id, comp_coder_t *comp_type, comp_info *c_info)
```

**sds_id**
IN: Data set identifier returned by **SDcreate** or **SDselect**

**comp_type**
OUT: Type of compression

**c_info**
OUT: Pointer to compression information structure

**Purpose**
Retrieves data set compression type and compression information.

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

**Description**
**SDgetcompress** retrieves the compression type and compression information for a data set.

The compression method is returned in the parameter **comp_type**. Valid values of **comp_type** are as follows:

- **COMP_CODE_NONE** (or 0) for no compression
- **COMP_CODE_RLE** (or 1) for RLE run-length encoding
- **COMP_CODE_NBIT** (or 2) for NBIT compression
- **COMP_CODE_SKPHUFF** (or 3) for Skipping Huffman compression
- **COMP_CODE_DEFLATE** (or 4) for GZIP compression

Additional compression method parameters are returned in the **c_info** struct in C and the array parameter **comp_prm** in Fortran. Note that **c_info** and **comp_prm** come into play only with compression modes that require additional parameters (i.e., other than **comp_type**); they are ignored in other cases.

The **c_info** struct is of type **comp_info**, contains algorithm-specific information for the library compression routines, and is described in the **hcomp.h** header file.

The **comp_prm** parameter is an array returning one or more parameters, as required by the compression method in use. Each compression parameter is returned as an element of the array, as follows:

- With Skipping Huffman compression, **comp_prm** is a 1-element array and **comp_prm(1)** contains the skip value, **skphuff_skp_size**.
- In the case of GZIP compression, **comp_prm** is also a 1-element array and **comp_prm(1)** contains the deflation value, **deflate_value**.
- In the case of NBIT compression, **comp_prm** is a 4-element array with **sign_ext** in **comp_prm(1)**, **fill_one** in **comp_prm(2)**, **start_bit** in **comp_prm(3)**, and **bit_len** in **comp_prm(4)**. **sign_ext**, **fill_one**, **start_bit**, and **bit_len** are discussed in the **SDsetnbitdataset/sfnbit** entry of the reference manual.

**FORTRAN**

```fortran
integer function sfgcompress(sds_id, comp_type, comp_prm)

integer sds_id, comp_type, comp_prm(*)
```
SDgetdatastrs/sfgdtstr

intn SDgetdatastrs(int32 sds_id, char *label, char *unit, char *format, char *coordsys, intn length)

sds_id IN: Data set identifier returned by SDcreate or SDselect
label OUT: Label (predefined attribute)
unit OUT: Unit (predefined attribute)
format OUT: Format (predefined attribute)
coordsys OUT: Coordinate system (predefined attribute)
length IN: Maximum length of the above predefined attributes

Purpose
Retrieves the predefined attributes of a data set.

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

Description
SDgetdatastrs retrieves the predefined attributes for the data set specified by the parameter sds_id. The predefined attributes are label, unit, format, and coordinate system. They are then stored in the parameters label, unit, format, and coordsys, respectively. Refer to Section 3.10 of the HDF User’s Guide for more information on predefined attributes.

If a particular data string is not stored, the first character of the corresponding SDgetdatastrs parameter is ‘\0’ in C. In FORTRAN, the parameter contains an empty string. Each string buffer must include the space to hold the null termination character. In C, if a user does not want a string back, NULL can be passed in for that string. Data strings are set by the SDsetdatastrs routine.

FORTRAN
integer function sfgdtstr(sds_id, label, unit, format, coordsys, length)

integer sds_id, length
character*(*) label, unit, format, coordsys
SDgetdimid/sfdimid

int32 SDgetdimid(int32 sds_id, intn dim_index)

\[ sds\_id \quad \text{IN: Data set identifier returned by SDcreate or SDselect} \]
\[ dim\_index \quad \text{IN: Index of the dimension} \]

\textbf{Purpose} \quad \text{Returns the identifier of a dimension given its index.}

\textbf{Return value} \quad \text{Returns the dimension identifier (dim_id) if successful and FAIL (or -1) otherwise.}

\textbf{Description} \quad \textbf{SDgetdimid} returns the identifier of the dimension specified by its index, the parameter \textit{dim_index}. The dimension index is a nonnegative integer and is less than the total number of data set dimensions returned by \textbf{SDgetinfo}.

\textbf{FORTRAN} \quad \text{integer function sfdimid(sds_id, dim_index)}

\quad \text{integer sds_id, dim_index}
**SDgetdimscale/sfgdscale**

```c
int n SDgetdimscale(int32 dim_id, VOIDP scale_buf)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dim_id</code></td>
<td>Dimension identifier returned by SDgetdimid</td>
</tr>
<tr>
<td><code>scale_buf</code></td>
<td>Buffer for the scale values</td>
</tr>
</tbody>
</table>

**Purpose**

Retrieves the scale values for a dimension.

**Return value**

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

**Description**

SDgetdimscale retrieves the scale values of the dimension identified by the parameter `dim_id` and stores the values in the buffer `scale_buf`.

SDdiminfo should be used to determine whether a scale has been set for the dimension, i.e., that the dimension scale data type is a valid HDF data type (not 0). Also use SDdiminfo to obtain the number of scale values for space allocation before calling SDgetdimscale.

It is not possible to read a subset of the scale values. SDgetdimscale returns all of the scale values stored with the given dimension.

The fact that SDgetdimscale returns SUCCEED should not be interpreted to mean that scale values have been defined for the data set. This function should always be used with SDdiminfo, which is used first to determine whether a scale has been set, the number of scale values, their data type, etc. If SDdiminfo indicates that no scale values have been set, the values returned by SDgetdimscale in data should be ignored.

**FORTRAN**

```fortran
integer function sfgdscale(dim_id, scale_buf)

integer dim_id

<valid numeric data type> scale_buf(*)
```
SDgetdimstrs/sfgdmstr

**Purpose**
Retrieves the predefined attributes of a dimension.

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

**Description**
**SDgetdimstrs** retrieves the predefined attributes associated with the dimension identified by the parameter `dim_id`. The predefined attributes are label, unit, and format. These predefined attributes are stored in the parameters `label`, `unit`, and `format`, respectively. Refer to Section 3.10 of the HDF User’s Guide for more information on predefined attributes.

If a particular data string was not stored, the first character of the corresponding **SDgetdimstrs** parameter is \0. Each string buffer must include space for the null termination character. If a user does not want a string returned, the corresponding parameter can be set to **NULL** in C and an empty string in Fortran. The predefined attributes are set by **SDsetdimstrs**.

**FORTRAN**
```fortran
integer function sfgdmstr(dim_id, label, unit, format, length)
  integer dim_id, length
  character*(*) label, unit, format
```
SDgetfillvalue/sfgfill/sfgcfill

intn SDgetfillvalue(int32 sds_id, VOIDP fill_value)

  sds_id        IN:  Data set identifier returned by SDcreate or SDselect
  fill_value    OUT:  Buffer for the returned fill value

Purpose
Reads the fill value of a data set, if the value has been set.

Return value
Returns SUCCEED (or 0) if a fill value is retrieved and FAIL (or -1) otherwise, including when the fill value is not set.

Description
SDgetfillvalue reads the fill value which has been set for the data set specified by the parameter sds_id. It is assumed that the data type of the fill value is the same as that of the data set.

Note that there are two FORTRAN-77 versions of this routine: sfgfill and sfgcfill. The sfgfill routine reads numeric fill value data and sfgcfill reads character fill value data.

FORTRAN
integer function sfgfill(sds_id, fill_value)
integer sds_id
<valid numeric data type> fill_value

integer function sfgcfill(sds_id, fill_value)
integer sds_id
character*(*) fill_value
SDgetinfo/sfginfo

intn SDgetinfo(int32 sds_id, char *sds_name, int32 rank, int32 dimsizes[], int32 *data_type, int32 *num_attrs)

sds_id  IN: Data set identifier returned by SDcreate and SDselect
sds_name OUT: Name of the data set
rank OUT: Number of dimensions in the data set
dimsizes OUT: Array containing the size of each dimension in the data set
data_type OUT: Data type for the data stored in the data set
num_attrs OUT: Number of attributes for the data set

Purpose
Retrieves the name, rank, dimension sizes, data type and number of attributes for a data set.

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

Description
SDgetinfo retrieves the name, number of dimensions, sizes of dimensions, data type, and number of attributes of the data set identified by sds_id, and stores them in the parameters sds_name, rank, dimsizes, data_type, and num_attrs, respectively.

The buffer sds_name can have at most 64 characters. If the name of the data set is not desired, then the parameter sds_name can be set to NULL in C and an empty string in Fortran.

The maximum value of the rank parameter is MAX_VAR_DIMS (or 32).

If the data set is created with an unlimited dimension, then in the C interface, the first element of the dimsizes array (corresponding to the slowest-changing dimension) contains the number of records in the unlimited dimension; in the FORTRAN-77 interface, the last element of the dimsizes array (corresponding to the slowest-changing dimension) contains this information. Use SDisrecord to determine if the data set has an unlimited dimension.

FORTRAN
integer function sfginfo(sds_id, sds_name, rank, dimsizes,
data_type, num_attrs)

character*(*) sds_name
integer sds_id, rank, dimsizes(*)
integer data_type, num_attrs
SDgetrange/sfgrange

SDgetrange/sfgrange

intn SDgetrange(int32 sds_id, VOIDP max, VOIDP min)

$sds_id$ IN: Data set identifier returned by SDcreate or SDselect
$max$ OUT: Maximum value of the range
$min$ OUT: Minimum value of the range

Purpose
Retrieves the maximum and minimum values of the range.

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

Description
SDgetrange retrieves the maximum value of the range into the parameter $max$ and the minimum value into the parameter $min$. The maximum and minimum values must be previously set via a call to SDsetrange.

It is assumed that the data type for the maximum and minimum range values are the same as that of the data.

FORTRAN

integer function sfgrange(sds_id, max, min)

integer sds_id
<valid numeric data type> max, min
**SDidtoref/sfid2ref**

```c
int32 SDidtoref(int32 sds_id)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sds_id</code></td>
<td>IN: Data set identifier returned by <strong>SDcreate</strong> or <strong>SDselect</strong></td>
</tr>
</tbody>
</table>

**Purpose**

Returns the reference number assigned to a data set.

**Return value**

Returns the data set reference number if successful and **FAIL** (or -1) otherwise.

**Description**

**SDidtoref** returns the reference number of the data set specified by the parameter `sds_id`. The reference number is assigned by the HDF library when the data set is created. The specified reference number can be used to add the data set to a vgroup as well as a means of using the HDF annotations interface to annotate the data set.

**FORTRAN**

```fortran
integer function sfid2ref(sds_id)

integer sds_id
```
SDiscoordvar/sfiscvar

intn SDiscoordvar(int32 sds_id)

\( sds_id \) \hspace{1cm} \text{IN: Data set identifier returned by \textbf{SDcreate} or \textbf{SDselect}}

**Purpose**
Determines if a data set is a coordinate variable.

**Return value**
Returns \texttt{TRUE} (or 1) if the data set is a coordinate variable, and \texttt{FALSE} (or 0) otherwise.

**Description**
\textbf{SDiscoordvar} determines if the data set specified by the parameter \texttt{sds_id} is a coordinate variable.

Coordinate variables are created to store metadata associated with dimensions. To ensure compatibility with netCDF, coordinate variables are implemented as data sets.

**FORTRAN**

\begin{verbatim}
integer function sfiscvar(sds_id)

integer sds_id
\end{verbatim}
### SDisdimval_bwcomp/sfisdmvc

```c
intn SDisdimval_bwcomp(int32 dim_id)
```

**dim_id**

IN: Dimension identifier returned by **SDgetdimid**

**Purpose**

Determines whether a dimension has the old and new representations or the new representation only.

Refer to the *HDF User’s Guide*, Chapter 3, titled *SD Scientific Data Sets (SD API)*, for information on old and new dimension representations.

**Return value**

Returns **SD_DIMVAL_BW_COMP** (or 1) if backward compatible, **SD_DIMVAL_BW_INCOMP** (or 0) if incompatible, **FAIL** (or -1) if error.

**Description**

**SDisdimval_bwcomp** will flag the dimension specified by the parameter `dim_id` as backward-compatible if a vdata with a class name of “DimVal0.0” does not exist in the vgroup for that dimension. If the vdata does exist, the specified dimension will be identified by **SDisdimval_bcomp** as backward-incompatible.

The compatibility mode can be changed by calls to **SDsetdimval_comp** at any time between the calls to **SDstart** and **SDend**.

```fortran
integer function sfisdmvc(dim_id)
```

integer `dim_id`
SDisrecord/sfisrcrd

int32 SDisrecord(int32 sds_id)

sds_id IN: Data set identifier returned by SDcreate or SDselect

Purpose Determines whether a data set is appendable.

Return value Returns TRUE (or 1) if the data set is appendable, and FALSE (or 0) otherwise.

Description SDisrecord will determine if the data set specified by the parameter sds_id is appendable, which means that the slowest-changing dimension was declared unlimited when the data set was created.

FORTRAN integer sfisrcrd(sd_id)

integer sd_id
**SDnametoindex/sfn2index**

```c
int32 SDnametoindex(int32 sd_id, char *sds_name)
```

- **sd_id**: IN: SD interface identifier returned by **SDstart**
- **sds_name**: IN: Name of the data set

**Purpose**
Determines the index of a data set given its name.

**Return value**
Returns the index of the data set (sds_index) if the data set is found and **FAIL** (or -1) otherwise.

**Description**
**SDnametoindex** returns the index of the data set with the name specified by the parameter **sds_name**. The routine does not accept wildcards in the specified data set name. It also searches on that name in a case-sensitive manner. If there are more than one data set with the same name, the routine will return the index of the first one.

**FORTRAN**
```fortran
integer function sfn2index(sd_id, sds_name)

integer sd_id
character*(*) sds_name
```
SDreadattr/sfrnatt/sfrcatt

intn SDreadattr(int32 obj_id, int32 attr_index, VOIDP attr_buf)

obj_id     IN:  Identifier of the object the attribute is attached to
attr_index IN:  Index of the attribute to be read
attr_buf   OUT:  Buffer for the attribute values

Purpose  Reads the values of an attribute.
Return value  Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.

Description  SDreadattr reads the values of the attribute specified by the parameter attr_index and stores the values in the buffer attr_buf. It is assumed that the user has called SDattrinfo to retrieve the number of attribute values and allocate sufficient space for the buffer. Note that the routine does not read a subset of attribute values.

The value of obj_id can be either an SD interface identifier (sd_id), returned by SDstart, a data set identifier (sds_id), returned by SDselect, or a dimension identifier (dim_id), returned by SDgetdimid.

The value of attr_index is a positive integer and is less than the total number of attributes. The index value can be obtained using the routines SDnametoindex and SDreftoindex. The total number of attributes for the object can be obtained using the routines SDgetinfo, SDattrinfo, SDdiminfo and SDfileinfo.

Note that this routine has two FORTRAN-77 versions: sfrnatt and sfrcatt. The sfrnatt routine reads numeric attribute data and sfrcatt reads character attribute data.

FORTRAN  

integer function sfrnatt(obj_id, attr_index, attr_buffer)

integer obj_id, attr_index
<valid numeric data> attr_buffer(*)

integer function sfrcatt(obj_id, attr_index, attr_buffer)

integer obj_id, attr_index
character*(*) attr_buffer
SDreadchunk/sfrchnk/sfrcchnk

intn SDreadchunk(int32 sds_id, int32 *origin, VOIDP datap)

*sds_id* IN: Data set identifier returned by SDcreate or SDselect

*origin* IN: Origin of the chunk to be read

*datap* OUT: Buffer for the chunk to be read

**Purpose**
Reads a data chunk from a chunked data set.

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

**Description**
SDreadchunk reads the entire chunk of data from the chunked data set identified by the parameter *sds_id*, and stores the data in the buffer *datap*. Reading starts at the location specified by the parameter *origin*. SDreadchunk is used when an entire chunk of data is to be read. SDreaddata is used when the read operation is to be done regardless of the chunking scheme used in the data set.

The parameter *origin* specifies the coordinates of the chunk according to the chunk position in the chunked array. Refer to the Chapter 3 of the HDF User's Guide, titled Scientific Data Sets (SD API), for a description of the organization of chunks in a data set.

SDreadchunk will return FAIL (or -1) when an attempt is made to read from a non-chunked data set.

Note that there are two FORTRAN-77 versions of this routine; one for numeric data (sfrchnk) and one for character data (sfrcchnk).

**FORTRAN**
integer sfrchnk(sds_id, origin, datap)

integer sds_id, origin(*)
<valid numeric data type> datap(*)

integer sfrcchnk(sds_id, origin, datap)

integer sds_id, origin(*)
character*(*) datap(*)
intn SDreaddata(int sds_id, int start[], int stride[], int edge[], VOIDP buffer)

sds_id IN: Data set identifier returned by SDcreate or SDselect
start IN: Array specifying the starting location from where data is read
stride IN: Array specifying the interval between the values that will be read along each dimension
edge IN: Array specifying the number of values to be read along each dimension
buffer OUT: Buffer to store the data read

Purpose
Reads a subsample of data from a data set or coordinate variable.

Return value
Returns SUCCEED (or 0) if successful or if the data set or coordinate variable contains no data and FAIL (or -1) otherwise.

Description
SDreaddata reads the specified subsample of data from the data set or coordinate variable identified by the parameter sds_id. The read data is stored in the buffer buffer. The subsample is defined by the parameters start, stride and edge.

The array start specifies the starting position from where the subsample will be read. Valid values of each element in the array start are from 0 to the size of the corresponding dimension of the data set - 1. The dimension sizes are returned by SDgetinfo.

The array edge specifies the number of values to read along each data set dimension.

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 data set will be read. If one of the elements of the array stride is 2, then every other element along the corresponding dimension of the data set will be read, and so on. Specifying stride value of NULL in the C interface or setting all values of the array stride to 1 in either interface specifies the contiguous reading of data. If all values in the array stride are set to 0, SDreaddata returns FAIL (or -1). No matter what stride value is provided, data is always placed contiguously in the buffer.

When reading data from a “chunked” data set using SDreaddata, consideration should be given to the issues presented in the section on chunking in Chapter 3 of the HDF User’s Manual, titled Scientific Data Sets (SD API) and Chapter 13 of the HDF User’s Manual, titled HDF Performance Issues.

Note that there are two FORTRAN-77 versions of this routine; sfrdata and sfrcdata. The sfrdata routine reads numeric scientific data and sfrcdata reads character scientific data.
Note

Regarding an important difference between the SD and GR interfaces:
The SD and GR interfaces differ in the correspondence between the dimension
order in parameter arrays such as start, stride, edge, and dimsizes and the
dimension order in the data array. See the SDreaddata and GRreadimage
reference manual pages for discussions of the SD and GR approaches,
respectively.

When writing applications or tools to manipulate both images and two-
dimensional SDs, this crucial difference between the interfaces must be taken
into account. While the underlying data is stored in row-major order in both
cases, the API parameters are not expressed in the same way. Consider the
example of an SD data set and GR image that are stored as identically-shaped
arrays of X columns by Y rows and accessed via the SDreaddata and
GRreadimage functions, respectively. Both functions take the parameters
start, stride, and edge:

- For SDreaddata, those parameters are expressed in \((y,x)\) or
  \([\text{row}, \text{column}]\) order. For example, \(\text{start}[0]\) is the starting point in the
  \(Y\) dimension and \(\text{start}[1]\) is the starting point in the \(X\) dimension. The
  same ordering holds true for all SD data set manipulation
  functions.

- For GRreadimage, those parameters are expressed in \((x,y)\) or
  \([\text{column}, \text{row}]\) order. For example, \(\text{start}[0]\) is the starting point in the
  \(X\) dimension and \(\text{start}[1]\) is the starting point in the \(Y\) dimension. The
  same ordering holds true for all GR functions manipulating image
data.

FORTRAN

```fortran
integer function sfrdata(sds_id, start, stride, edge, buffer)
integer sds_id, start(*), stride(*), edge(*)
<valid numeric data type> buffer(*)

integer function sfrcdata(sds_id, start, stride, edge, buffer)
integer sds_id, start(*), stride(*), edge(*)
character(*) buffer
```
**SDreftoindex/sfref2index**

```
int32 SDreftoindex(int32 sd_id, int32 sds_ref)
```

- **sd_id** (IN): SD interface identifier returned by **SDstart**
- **sds_ref** (IN): Reference number of the data set

**Purpose**

Returns the index of a data set given the reference number.

**Return value**

Returns the index of the data set (**sds_index**) if the data set is found and **FAIL** (or -1) otherwise.

**Description**

**SDreftoindex** returns the index of a data set identified by its reference number, **sds_ref**.

The value of **sds_index** returned by **SDreftoindex** can be passed to **SDselect** to obtain a data set identifier (**sds_id**).

**FORTRAN**

```
integer function sfref2index(sd_id, sds_ref)
```

```
integer sd_id, sds_ref
```
SDselect/sfselect

int32 SDselect(int32 sd_id, int32 sds_index)

sd_id IN: SD interface identifier returned by SDstart
sds_index IN: Index of the data set

Purpose Obtains the data set identifier (sds_id) of a data set.
Return value Returns the data set identifier (sds_id) if successful and FAIL (or -1) otherwise.
Description SDselect obtains the data set identifier (sds_id) of the data set specified by its index, sds_index.

The integration with netCDF has required that a dimension (or coordinate variable) is stored as a data set in the file. Therefore, the value of sds_index may correspond to the coordinate variable instead of the actual data set. Users should use the routine SDcoordvar to determine whether the given data set is a coordinate variable.

The value of sds_index is greater than or equal to 0 and less than the number of data sets in the file. The total number of data sets in a file may be obtained from a call to SDfileinfo. The SDnametoindex routine can be used to find the index of a data set if its name is known.

Fortran integer function sfselect(sd_id, sds_index)
integer sd_id, sds_index
Sfssnatt/sfsnatt/sfscatt

intn Sfssnatt(int32 obj_id, char *attr_name, int32 data_type, int32 count, VOIDP values)

- **obj_id** (IN): Identifier of the object the attribute is to be attached to.
- **attr_name** (IN): Name of the attribute.
- **data_type** (IN): Data type of the values in the attribute.
- **count** (IN): Total number of values to be stored in the attribute.
- **values** (IN): Data values to be stored in the attribute.

**Purpose**: Attaches an attribute to an object.

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

**Description**: **Sfssnatt** attaches the attribute to the object specified by the `obj_id` parameter. The attribute is defined by its name, `attr_name`, data type, `data_type`, number of attribute values, `count`, and the attribute values, `values`. **Sfssnatt** provides a generic way for users to define metadata. It implements the label = value data abstraction.

The value of `obj_id` can be an SD interface identifier (sd_id), returned by **SDcreate**, a data set identifier (sds_id), returned by **SDselect**, or a dimension identifier (dim_id), returned by **SDgetdimid**.

If an SD interface identifier (sd_id) is specified as the `obj_id` parameter, a global attribute is created which applies to all objects in the file. If a data set identifier (sds_id) is specified as the `obj_id` parameter, an attribute is attached to the specified data set. If a dimension identifier (dim_id) is specified as the `obj_id` parameter, an attribute is attached to the specified dimension.

The `attr_name` argument can be any ASCII string.

The `data_type` parameter can contain any data type supported by the HDF library. These data types are listed in Table 1A in Section 1 of this manual.

Attribute values are passed in the parameter `values`. The number of attribute values is defined by the `count` parameter. 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.

Note that there are two FORTRAN-77 versions of this routine; **sfsnatt** and **sfscatt**. The **sfsnatt** routine writes numeric attribute data and **sfscatt** writes character attribute data.

**FORTRAN**

```fortran
integer function sfsnatt(obj_id, attr_name, data_type, count, values)

integer obj_id, data_type, count
character*(*) attr_name
```
<valid numeric data type> values(*)

integer function sfscatt(obj_id, attr_name, data_type, count, values)

integer obj_id, data_type, count
character(*) attr_name, values
**SDsetblocksize/sfsblsz**

```c
intn SDsetblocksize(int32 sd_id, int32 block_size)
```

- **sd_id** IN: SD interface identifier returned by SDstart
- **block_size** IN: Size of the block in bytes

**Purpose**
Sets the block size used for storing data sets with unlimited dimensions.

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

**Description**
SDsetblocksize sets the block size defined in the parameter block_size for all data sets in the file. SDsetblocksize is used when creating new data sets only; it has no effect on pre-existing data sets.

SDsetblocksize must be used after calls to SDcreate or SDselect and before the call to SDwritedata.

The block_size parameter should be set to a multiple of the desired buffer size.

**FORTRAN**
```
integer sfsblsz(sd_id, block_size)
```
```
integer sd_id, block_size
```
### SDsetcal/sfscal

#### Purpose
Sets the calibration information.

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

#### Description
SDsetcal stores the calibration record associated with a data set. A calibration record contains the following information:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sds_id</td>
<td>IN: Data set identifier returned by SDcreate or SDselect</td>
</tr>
<tr>
<td>cal</td>
<td>IN: Calibration factor</td>
</tr>
<tr>
<td>cal_err</td>
<td>IN: Calibration error</td>
</tr>
<tr>
<td>offset</td>
<td>IN: Uncalibrated offset</td>
</tr>
<tr>
<td>offset_err</td>
<td>IN: Uncalibrated offset error</td>
</tr>
<tr>
<td>data_type</td>
<td>IN: Data type of uncalibrated data</td>
</tr>
</tbody>
</table>

The relationship between a value cal_value stored in a data set and the original value is defined as: orig_value = cal \* (cal_value - offset).

The variable offset_err contains a potential error of offset, and cal_err contains a potential error of cal. Currently the calibration record is provided for information only. The SD interface performs no operations on the data based on the calibration tag.

The calibration information is automatically cleared after a call to SDreaddata or SDwritedata. Therefore, SDsetcal must be called once for each data set that is to be read or written.

#### FORTRAN

```fortran
integer function sfscal(sds_id, cal, cal_err, offset, offset_err, data_type)

integer sds_id, data_type
real*8 cal, cal_err, offset, offset_err
```


SDsetchunk/sfschnk

**Purpose**

Sets the chunk size and the compression method, if any, of a data set.

**Return value**

Returns `SUCCEED` (or 0) if successful and `FAIL` (or -1) otherwise.

**Description**

`SDsetchunk` makes the data set specified by the parameter `sds_id` a chunked data set according to the chunking and compression information provided in the parameters `cdef` and `flag` in C, and in the parameters `comp_type` and `comp_prm` in Fortran.

**C only:**

The parameter `flag` specifies the type of the data set, i.e., if the data set is chunked or chunked and compressed with either RLE, Skipping Huffman, GZIP, Szip, or NBIT compression methods. Valid values of `flag` are `HDF_CHUNK` for a chunked data set, `HDF_CHUNK | HDF_COMP` for a chunked data set compressed with RLE, Skipping Huffman, Szip, or GZIP compression methods, and `HDF_CHUNK | HDF_NBIT` for a chunked NBIT-compressed data set.

Chunking and compression information is passed in the parameter `cdef`. The parameter `cdef` has a type of `HDF CHUNK_DEF`, defined in the HDF library as follows:
typedef union hdf_chunk_def_u
{
    int32 chunk_lengths[2];    /* chunk lengths along each dim */

    struct
    {
        int32 chunk_lengths[2];
        int32 comp_type;        /* compression type */
        struct comp_info cinfo;
    } comp;

    struct
    {
        int32 chunk_lengths[2];
        intn start_bit;
        intn bit_len;
        intn sign_ext;
        intn fill_one;
    } nbit;
} HDF_CHUNK_DEF

There are three pieces of chunking and compression information which should be specified: chunking dimensions, compression type, and, if needed, compression parameters.

If the data set is chunked, i.e., flag value is HDF_CHUNK, then chunk_lengths[] elements of cdef union (cdef.chunk_lengths[]) have to be initialized to the chunk dimensions.

If the data set is chunked and compressed using RLE, Skipping Huffman or GZIP methods (i.e., flag value is set up to HDF_CHUNK | HDF_COMP), then the elements chunk_lengths[] of the structure comp in the union cdef (cdef.comp.chunk_lengths[]) have to be initialized to the chunk dimensions.

If the data set is chunked and NBIT compression is applied (i.e., flag values is set up to HDF_CHUNK | HDF_NBIT), then the elements chunk_lengths[] of the structure nbit in the union cdef (cdef.nbit.chunk_lengths[]) have to be initialized to the chunk dimensions.

Compression types are passed in the field comp_type of the structure cinfo, which is an element of the structure comp in the union cdef (cdef.comp.cinfo.comp_type). Valid compression types are: COMP_CODE_RLE for RLE, COMP_CODE_SKPHUFF for Skipping Huffman, COMP_CODE_SZIP for Szip compression, and COMP_CODE_DEFLATE for GZIP compression.

For Skipping Huffman and GZIP compression parameters are passed in corresponding fields of the structure cinfo. Specify skipping size for Skipping Huffman compression in the field cdef.comp.cinfo.skphuff.skp_size; this value must be 1 or greater. Specify the deflate level for GZIP compression in the field cdef.comp.cinfo.deflate_level. Valid deflate level values are integers between 0 and 9 inclusive.

For Szip compression, parameters must be passed in a structure identical to the structure described in the SDsetcompress reference manual entry.

Refer to the SDsetcompress page in this manual for the definition of the structure comp_info.

NBIT compression parameters are specified in the fields start_bit, bit_len, sign_ext, and fill_one in the structure nbit of the union cdef.
**Fortran only:**

The `dim_length` array specifies the chunk dimensions.

The `comp_type` parameter specifies the compression type. Valid compression types and their values are defined in the `hdf.inc` file, and are listed below.

- `COMP_CODE_NONE` (or 0) for uncompressed data
- `COMP_CODE_RLE` (or 1) for data compressed using the RLE compression algorithm
- `COMP_CODE_NBIT` (or 2) for data compressed using the NBIT compression algorithm
- `COMP_CODE_SKPHUFF` (or 3) for data compressed using the Skipping Huffman compression algorithm
- `COMP_CODE_DEFLATE` (or 4) for data compressed using the GZIP compression algorithm
- `COMP_CODE_SZIP` (or 5) for data compressed using the Szip compression algorithm

The `comp_prm(1)` parameter specifies the skipping size for the Skipping Huffman compression method and the deflate level for the GZIP compression method. The skipping size value must be 1 or greater; the deflate level must be an integer value between 0 and 9 inclusive.

For NBIT compression, the four elements of the array `comp_prm` correspond to the four NBIT compression parameters listed in the structure `nbit`. The value of `comp_prm(1)` should be set to the value of `start_bit`, the value of `comp_prm(2)` should be set to the value of `bit_len`, the value of `comp_prm(3)` should be set to the value of `sign_ext`, and the value of `comp_prm(4)` should be set to the value of `fill_one`. See the `HDF_CHUNK_DEF` union description and the description of `SDsetnbitdataset` function for NBIT compression parameters definitions.

**FORTRAN**

```fortran
integer sfschnk(sds_id, dim_length, comp_type, comp_prm)
integer sds_id, dim_length, comp_type, comp_prm(*)
```
**SDsetchunkcache/sfschhnk**

```c
intn SDsetchunkcache(int32 sds_id, int32 maxcache, int32 flag)
```

- **sds_id**: IN: Data set identifier returned by `SDcreate` or `SDselect`
- **maxcache**: IN: Maximum number of chunks in the cache
- **flag**: IN: Flag determining the behavior of the routine

**Purpose**: Sets the size of the chunk cache.

**Return value**: Returns the maximum number of chunks that can be cached (the value of the parameter `maxcache`) if successful and `FAIL` (or `-1`) otherwise.

**Description**:
- `SDsetchunkcache` sets the size of the chunk cache to the value of the parameter `maxcache`.

Currently the only allowed value of the parameter `flag` is 0, which designates default operation.

By default, when a generic data set is promoted to be a chunked data set, the parameter `maxcache` is set to the number of chunks along the fastest changing dimension and a cache for the chunks is created.

If the chunk cache is full and the value of the parameter `maxcache` is greater than the current `maxcache` value, then the chunk cache is reset to the new value of `maxcache`. Otherwise the chunk cache remains at the current value of `maxcache`. If the chunk cache is not full, then the chunk cache is set to the new value of `maxcache` only if the new `maxcache` value is greater than the current number of chunks in the cache.

Do not set the value of `maxcache` to be less than the number of chunks along the fastest-changing dimension of the biggest slab to be written or read via `SDreaddata` or `SDwritedata`. Doing this will cause internal thrashing. See the section on chunking in Chapter 13, "HDF Performance Issues," in the *HDF User’s Guide*, for more information on this.

**FORTRAN**

```fortran
integer sfscchnk(sds_id, maxcache, flag)
```

```fortran
integer sds_id, maxcache, flag
```
SDsetcompress/sfscompress

intn SDsetcompress(int32 sds_id, int32 comp_type, comp_info *c_info)

sds_id
IN: Data set identifier returned by SDcreate or SDselect

comp_type IN: Compression method

C only:
c_info
IN: Pointer to the comp_info union

Fortran only:
comp_prm
IN: Compression parameters array

Purpose
Compresses the data set with the specified compression method.

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

Description SDsetcompress compresses the data set identified by the parameter sds_id according to the compression method specified by the parameter comp_type and the compression information specified by the parameter c_info in C and comp_prm in Fortran. SDsetcompress compresses the data set data at the time it is called, not during the next call to SDwritedata.

SDsetcompress is a simplified interface to the HCcreate routine and should be used instead of HCcreate unless the user is familiar with working with the lower-level routines.

The parameter comp_type is the compression type definition and is set to COMP_CODE_RLE (or 1) for run-length encoding (RLE), COMP_CODE_SKPHUFF (or 3) for Skipping Huffman, COMP_CODE_DEFLATE (or 4) for GZIP compression, COMP_CODE_SZIP (or 5) for Szip compression, or COMP_CODE_NONE (or 0) for no compression.

The parameter c_info is a pointer to a union structure of type comp_info. This union structure is defined as follows:
typedef union tag_comp_info
{
  struct
  {
    /* Not used by SDsetcompress */
    jpeg;
  } jpeg;

  struct
  {
    /* Not used by SDsetcompress */
    nbit;
  } nbit;

  struct
  {
    /* struct to contain info about how to compress size of the
     elements when skipping */
    int n_skp_size;
  } skphuff;

  struct
  {
    /* struct to contain info about how to compress or decompress
     gzip encoded dataset how hard to work when compressing
     data */
    int n_level;
  } deflate;

  struct
  {
    /* struct to contain info about how to compress or decompress
     szip encoded dataset */
    int32 bits_per_pixel;
    int32 compression_mode;
    int32 options_mask;
    int32 pixels;
    int32 pixels_per_block;
    int32 pixels_per_scanline;
  } szip;
} comp_info;

The skipping size for the Skipping Huffman algorithm must be 1 or greater and
is specified in the field c_info.skphuff.skp_size in C and in the parameter
comp_prm(1) in Fortran.

The deflate level for the GZIP algorithm is specified in the
c_info.deflate.level field in C and in the parameter comp_prm(1) in
Fortran. Valid values are integers between 0 and 9 inclusive.

FORTRAN

integer sfscompress(sds_id, comp_type, comp_prm)

integer sds_id, comp_type, comp_prm(*)
SDsetdatastrs/sfsdtstr

SDsetdatastrs/sfsdtstr

```
intn SDsetdatastrs(int32 sds_id, char *label, char *unit, char *format, char *coordsys)
```

- **sds_id**: IN: Data set identifier returned by SDcreate or SDselect.
- **label**: IN: Label (predefined attribute).
- **unit**: IN: Unit (predefined attribute).
- **format**: IN: Format (predefined attribute).
- **coordsys**: IN: Coordinate system (predefined attribute).

**Purpose**: Sets the predefined attributes for a data set.

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

**Description**: SDsetdatastrs sets the predefined attributes of the data set, identified by `sds_id`, to the values specified in the parameters `label`, `unit`, `format` and `coordsys`. The predefined attributes are label, unit, format, and coordinate system. If the user does not want a string returned, the corresponding parameter can be set to NULL in C and an empty string in Fortran.

For more information about predefined attributes, refer to Section 3.10 of the HDF User’s Guide.

**FORTRAN**

```
integer function sfsdtstr(sds_id, label, unit, format, coordsys)

integer sds_id
character(*) label, unit, format, coordsys
```
**SDsetdimname/sfsetdimname**

```c
intn SDsetdimname(int32 dim_id, char *dim_name)
```

**dim_id**

IN: Dimension identifier returned by `SDgetdimid`

**dim_name**

IN: Name of the dimension

**Purpose**

Assigns a name to a dimension.

**Return value**

Returns `SUCCEED` (or 0) if successful and `FAIL` (or -1) otherwise.

**Description**

`SDsetdimname` sets the name of the dimension identified by the parameter `dim_id` to the value specified in the parameter `dim_name`. Dimensions that are not explicitly named by the user will have the default name of “fakeDim[x]” specified by the HDF library, where [x] denotes the dimension index.

If another dimension exists with the same name it is assumed that they refer to the same dimension object and changes to one will be reflected in the other. If the dimension with the same name has a different size, an error condition will result.

Naming dimensions is optional but encouraged.

The length of the parameter `dim_name` can be at most 64 characters.

**FORTRAN**

```fortran
integer function sfsetdimname(dim_id, dim_name)

integer dim_id
character*(*) dim_name
```
**SDsetdimscale/sfsdscale**

\[
\text{intn SDsetdimscale(int32 \textit{dim\_id}, \textit{count}, \textit{int32 data\_type}, \text{VOIDP data})}
\]

- **\textit{dim\_id}** IN: Dimension identifier returned by **SDgetdimid**
- **\textit{count}** IN: Total number of values along the dimension
- **\textit{data\_type}** IN: Data type of the values along the dimension
- **\textit{data}** IN: Value of each increment along the dimension

**Purpose**
Stores the values of a dimension.

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

**Description**
\textbf{SDsetdimscale} stores scale information for the dimension identified by the parameter \textit{dim\_id}. Note that it is possible to store dimension scale values without naming the dimension.

For fixed-size arrays, the value of \textit{count} must be equal to the the dimension size or the routine will fail.

Note that, due to the existence of the parameter \textit{data\_type}, the dimension scales need not have the same data type as the data set.

Note that if \textbf{SDsetdimscale} is called and \textbf{SDsetdimname} is subsequently called for the same dimension, \textbf{SDsetdimscale} must be called again to reassociate the scale with the new name.

**FORTRAN**
\[
\text{integer function sfsdscale(dim\_id, count, data\_type, data)}
\]

\[
\text{integer dim\_id, count, data\_type}
\]

\[
<\text{valid data type}> \text{data(*)}
\]
**SDsetdimstrs/sfsdmstr**

```c
intn SDsetdimstrs(int32 dim_id, char *label, char *unit, char *format)
```

- **dim_id** IN: Dimension identifier returned by **SDgetdimid**
- **label** IN: Label (predefined attribute)
- **unit** IN: Unit (predefined attribute)
- **format** IN: Format (predefined attribute)

**Purpose**
Sets the predefined attribute of a dimension.

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

**Description**
**SDsetdimstrs** sets the predefined attribute (label, unit, and format) for a dimension and its scale to the values specified in the parameters `label`, `unit` and `format`. If a parameter is set to NULL in C and an empty string in Fortran, then the attribute corresponding to that parameter will not be written. For more information about predefined attributes, refer to Section 3.10 of the HDF User’s Guide.

**FORTRAN**
```fortran
integer function sfsdmstr(dim_id, label, unit, format)
```

```fortran
integer dim_id
character*(*) label, unit, format
```
### SDsetdimval_comp/sfsdmvc

**intn SDsetdimval_comp(int32 dim_id, intn comp_mode)**

- **dim_id**  
  IN: Dimension identifier returned by **SDgetdimid**

- **comp_mode**  
  IN: Compatibility mode to be set

**Purpose**  
Determines whether a dimension will have the old and new representations or the new representation only.

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

**Description**  
**SDsetdimval_comp** sets the compatibility mode specified by the **comp_mode** parameter for the dimension identified by the **dim_id** parameter. The two possible compatibility modes are: “backward-compatible” mode, which implies that the old and new dimension representations are written to the file, and “backward-incompatible” mode, which implies that only the new dimension representation is written to the file.

Unlimited dimensions are always backward-compatible, therefore **SDsetdimval_comp** takes no action on unlimited dimensions.

As of HDF version 4.1r1, the default mode is backward-incompatible. Subsequent calls to **SDsetdimval_comp** will override the settings established in previous calls to the routine.

The **comp_mode** parameter can be set to **SD_DIMVAL_BW_COMP** (or 1), which specifies backward-compatible mode, or **SD_DIMVAL_BW_INCOMP** (or 0), which specifies backward-incompatible mode.

**FORTRAN**  
integer function sfsdmvc(dim_id, comp_mode)

integer dim_id, comp_mode
### SDsetexternalfile/sfsextf

**Function**

```c
intn SDsetexternalfile(int32 sds_id, char *filename, int32 offset)
```

**IN**:

- `sds_id`:
  Data set identifier returned by `SDcreate` or `SDselect`
- `filename`:
  Name of the external file
- `offset`:
  Number of bytes from the beginning of the external file to where the data will be written

**Purpose**

Stores data in an external file.

**Return value**

Returns `SUCCEED` (or 0) if successful and `FAIL` (or -1) otherwise.

**Description**

`SDsetexternalfile` allows users to move the actual data values (i.e., not metadata) of a data set, `sds_id`, into the external data file named by the parameter `filename`, and started at the offset specified by the parameter `offset`. The metadata remains in the original file. Note that this routine works only with HDF post-version 3.2 files.

Data can only be moved once for any given data set, and it is the user's responsibility to make sure the external data file is kept with the "original" file.

If the data set already exists, its data will be moved to the external file. Space occupied by the data in the primary file will not be released. To release the space in the primary file use the `hdfpack` command-line utility. If the data set does not exist, its data will be written to the external file during the consequent calls to `SDwritedata`.

See the Reference Manual entries for `HXsetcreatedir` and `HXsetdir` for more information on the options available for accessing external files.

**FORTRAN**

```fortran
integer function sfsextf(sds_id, file_name, offset)

integer sds_id, offset
character(*) file_name
```
SDsetfillmode/sfsflmd

intn SDsetfillmode(int32 sd_id, intn fill_mode)

sd_id  IN:   SD interface identifier returned by SDstart
fill_mode  IN:   Fill mode

Purpose  Sets the current fill mode of a file.

Return value  Returns the fill mode value before it was reset if successful and FAIL (or -1) otherwise.

Description  SDsetfillmode applies the fill mode specified by the parameter fill_mode to all data sets contained in the file identified by the parameter sd_id.

Possible values of fill_mode are SD_FILL (or 0) and SD_NOFILL (or 256). SD_FILL is the default mode, and indicates that fill values will be written when the data set is created. SD_NOFILL indicates that fill values will not be written.

When a data set without unlimited dimensions is created, by default the first SDwritedata call will fill the entire data set with the default or user-defined fill value (set by SDsetfillvalue). In data sets with an unlimited dimension, if a new write operation takes place along the unlimited dimension beyond the last location of the previous write operation, the array locations between these written areas will be initialized to the user-defined fill value, or the default fill value if a user-defined fill value has not been specified.

If it is certain that all data set values will be written before any read operation takes place, there is no need to write the fill values. Simply call SDsetfillmode with fill_mode value set to SD_NOFILL, which will eliminate all fill value write operations to the data set. For large data sets, this can improve the speed by almost 50%.

FORTRAN  integer function sfsflmd(sd_id, fill_mode)

integer sd_id, fill_mode
**SDsetfillvalue/sfsfill/sfscfill**

```c
int SDsetfillvalue(int32 sds_id, VOIDP fill_value)
```

- `sds_id` IN: Data set identifier returned by **SDcreate** or **SDselect**
- `fill_value` IN: Fill value

**Purpose** Sets the fill value for a data set.

**Return value** Returns succeed (or 0) if successful and fail (or -1) otherwise.

**Description** **SDsetfillvalue** sets the fill value specified by the `fill_value` parameter for the data set identified by the `sds_id` parameter.

The fill value is assumed to have the same data type as the data set.

It is recommended to call **SDsetfillvalue** before writing data.

**FORTRAN**

```fortran
integer function sfsfill(sds_id, fill_value)

integer sds_id
<valid numeric data type> fill_value

integer function sfscfill(sds_id, fill_value)

integer sds_id
character*(*) fill_value
```
SDsetnbitdataset/sfsnbit

intn SDsetnbitdataset(int32 sds_id, intn start_bit, intn bit_len, intn sign_ext, intn fill_one)

sds_id IN: Data set identifier returned by SDcreate or SDselect
start_bit IN: Leftmost bit of the field to be written
bit_len IN: Length of the bit field to be written
sign_ext IN: Sign extend specifier
fill_one IN: Background bit specifier

Purpose
Specifies a non-standard bit length for the data set values.

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

Description
SDsetnbitdataset allows the HDF user to specify that the data set identified by the parameter sds_id contains data of a non-standard length defined by the parameters start_bit and bit_len. Additional information about the non-standard bit length decoding are specified in the parameters sign_ext and fill_one.

Any length between 1 and 32 bits can be specified. After SDsetnbitdataset has been called for the data set array, any read or write operations will involve a conversion between the new data length of the data set array and the data length of the read or write buffer.

Bit lengths of all data types are counted from the right of the bit field starting with 0. In a bit field containing the values 01111011, bits 2 and 7 are set to 0 and all the other bits are set to 1.

The start_bit parameter specifies the leftmost position of the variable-length bit field to be written. For example, in the bit field described in the preceding paragraph a start_bit parameter set to 4 would correspond to the fourth bit value of 1 from the right.

The bit_len parameter specifies the number of bits of the variable-length bit field to be written. This number includes the starting bit and the count proceeds toward the right end of the bit field - toward the lower-bit numbers. For example, starting at bit 5 and writing 4 bits of the bit field described in the preceding paragraph would result in the bit field 1110 being written to the data set. This would correspond to a start_bit value of 5 and a bit_len value of 4.

The sign_ext parameter specifies whether to use the leftmost bit of the variable-length bit field to sign-extend to the leftmost bit of the data set data. For example, if 9-bit signed integer data is extracted from bits 17-25 and the bit in position 25 is 1, then when the data is read back from disk, bits 26-31 will be set to 1. Otherwise bit 25 will be 0 and bits 26-31 will be set to 0. The sign_ext parameter can be set to TRUE (or 1) or FALSE (or 0) - specify TRUE to sign-extend.

The fill_one specifies whether to fill the “background” bits with the value 1 or 0. This parameter can also be set to TRUE or FALSE.
The “background” bits of a variable-length data set are the bits that fall outside of the variable-length bit field stored on disk. For example, if five bits of an unsigned 16-bit integer data set located in bits 5 to 9 are written to disk with the fill_one parameter set to true (or 1), then when the data is reread into memory bits 0 to 4 and 10 to 15 would be set to 1. If the same 5-bit data was written with a fill_one value of false (or 0), then bits 0 to 4 and 10 to 15 would be set to 0.

This bit operation is performed before the sign-extend bit-filling. For example, using the sign_ext example above, bits 0 to 16 and 26 to 31 will first be set to the “background” bit value, and then bits 26 to 31 will be set to 1 or 0 based on the value of the 25th bit.

**FORTRAN**

```fortran
integer function sfsnbit(sds_id, start_bit, bit_len, sign_ext, 
                       fill_one)

integer sds_id, start_bit, bit_len, sign_ext, fill_one
```


SDsetrange/sfsrange

SDsetrange/sfsrange

intn SDsetrange(int32 sds_id, VOIDP max, VOIDP min)

$sds_id$ IN: Data set identifier returned by SDcreate or SDselect
$max$ IN: Maximum value of the range
$min$ IN: Minimum value of the range

Purpose
Sets the maximum and minimum range values for a data set.

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

Description
SDsetrange sets the maximum and minimum range values of the data set identified by the parameter sds_id with the values of the parameters max and min. The term “range” is used here to describe the range of numeric values stored in a data set.

It is assumed that the data type for the maximum and minimum range values are the same as the data type of the data.

This routine does not compute the maximum and minimum range values, it only stores the values as given. As a result, the maximum and minimum range values may not always reflect the actual maximum and minimum range values in the data set data.

FORTRAN
integer function sfsrange(sds_id, max, min)

integer sds_id

<valid numeric data type> max, min
**SDstart/sfstart**

```c
int32 SDstart(char *filename, int32 access_mode)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>filename</code></td>
<td>Name of the HDF file</td>
</tr>
<tr>
<td><code>access_mode</code></td>
<td>The file access mode in effect during the current session</td>
</tr>
</tbody>
</table>

**Purpose**

Opens an HDF file and initializes an SD interface.

**Return value**

Returns an SD interface identifier if successful and `FAIL` (or -1) otherwise.

**Description**

`SDstart` opens the file with the name specified by the parameter `filename`, with the access mode specified by the parameter `access_mode`, and returns an SD interface identifier (`sd_id`). This routine must be called for each file before any other SD calls can be made on that file.

The type of identifier returned by `SDstart` is currently not the same as the identifier returned by `Hopen`. As a result, the SD interface identifiers (`sd_id`) returned by this routine are not understood by other HDF interfaces.

To mix SD API calls and other HDF API calls, use `SDstart` and `Hopen` on the same file. `SDstart` must precede all SD calls, and `Hopen` must precede all other HDF function calls. To terminate access to the file, use `SDend` to dispose of the SD interface identifier, `sd_id`, and `Hclose` to dispose of the file identifier, `file_id`.

The file identified by the parameter `filename` can be any one of the following: an XDR-based netCDF file, “old-style” DFSD file or a “new-style” SD file.

The value of the parameter `access_mode` can be one of the following:

- **DFACC_READ** - Open existing file for read-only access. If the file does not exist, specifying this mode will cause `SDstart` to return `FAIL` (or -1).
- **DFACC_WRITE** - Open existing file for read and write access. If the file does not exist, specifying this mode will cause `SDstart` to return `FAIL` (or -1).
- **DFACC_CREATE** - Create a new file with read and write access. If the file has already existed, its contents will be replaced.

**FORTRAN**

```fortran
integer function sfstart(filename, access_mode)

character(*) filename
integer access_mode
```

December 30, 2003 2-137
intn SDwritechunk(int32 sds_id, int32 *origin, VOIDP datap)

**sds_id**
IN: Data set identifier returned by **SDcreate** or **SDselect**

**origin**
IN: Origin of the chunk to be written

**datap**
IN: Buffer for the chunk data to be written

**Purpose**
Writes a data chunk to a chunked data set.

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

**Description**
**SDwritechunk** writes the entire chunk of data stored in the buffer **datap** to the chunked data set identified by the parameter **sds_id**. Writing starts at the location specified by the parameter **origin**. **SDwritechunk** is used when an entire chunk of data is to be written. **SDwritedata** is used when the write operation is to be done regardless of the chunking scheme used in the data set.

**SDwritechunk** will return **FAIL** (or -1) when an attempt is made to use it to write to a non-chunked data set.

The parameter **origin** specifies the coordinates of the chunk according to the chunk position in the overall chunk array. Refer to Chapter 3 of the HDF User’s Guide, titled *Scientific Data Sets (SD API)*, for a description of the organization of chunks in a data set.

Note that there are two FORTRAN-77 versions of this routine; one for numeric data (**sfwchnk**) and one for character data (**sfwcchnk**).

**FORTRAN**

```fortran
integer sfwchnk(sds_id, origin, datap)

integer sds_id, origin
<valid numeric data type> datap(*)

integer sfwcchnk(sds_id, origin, datap)

integer sds_id, origin
character*(*) datap(*)
```
SDwritedata/sfwdata/sfwcdata

intn SDwritedata(int32 sds_id, int32 start[], int32 stride[], int32 edge[], VOIDP buffer)

  sds_id  IN: Data set identifier returned by SDcreate or SDselect
  start   IN: Array specifying the starting location of the data to be written
  stride  IN: Array specifying the number of values to skip along each dimension
  edge    IN: Array specifying the number of values to be written along each dimension
  buffer  IN: Buffer for the values to be written

Purpose
Writes a subsample of data to a data set or to a coordinate variable.

Return value
Returns succeed (or 0) if successful and fail (or -1) otherwise.

Description
SDwritedata writes the specified subsample of data to the data set or coordinate variable identified by the parameter sds_id. The data is written from the buffer buffer. The subsample is defined by the parameters start, stride and edge.

The array start specifies the starting position from where the subsample will be written. Valid values of each element in the array start are from 0 to the size of the corresponding dimension of the data set - 1. The dimension sizes are returned by SDgetinfo.

The array edge specifies the number of values to write along each data set dimension.

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 data set will be written. If one of the elements of the array stride is 2, then every other element along the corresponding dimension of the data set will be written, and so on. Specifying stride value of null in the C interface or setting all values of the array stride to 1 in either interface specifies the contiguous writing of data. If all values in the array stride are set to 0, SDwritedata returns fail (or -1).

When writing data to a chunked data set using SDwritedata, consideration should be given to be issues presented in the section on chunking in Chapter 3 of the HDF User's Manual, titled Scientific Data Sets (SD API) and Chapter 13 of the HDF User's Manual, titled HDF Performance Issues.

Note that there are two FORTRAN-77 versions of this routine; sfwdata and swcdata. The sfwdata routine writes numeric data and swcdata writes character scientific data.
Note

Regarding an important difference between the SD and GR interfaces:
The SD and GR interfaces differ in the correspondence between the dimension order in parameter arrays such as start, stride, edge, and dimsizes and the dimension order in the data array. See the SDreaddata and GRreadimage reference manual pages for discussions of the SD and GR approaches, respectively.

When writing applications or tools to manipulate both images and two-dimensional SDs, this crucial difference between the interfaces must be taken into account. While the underlying data is stored in row-major order in both cases, the API parameters are not expressed in the same way. Consider the example of an SD data set and GR image that are stored as identically-shaped arrays of X columns by Y rows and accessed via the SDreaddata and GRreadimage functions, respectively. Both functions take the parameters start, stride, and edge.

- For SDreaddata, those parameters are expressed in (y, x) or [row, column] order. For example, start[0] is the starting point in the Y dimension and start[1] is the starting point in the X dimension. The same ordering holds true for all SD data set manipulation functions.

- For GRreadimage, those parameters are expressed in (x, y) or [column, row] order. For example, start[0] is the starting point in the X dimension and start[1] is the starting point in the Y dimension. The same ordering holds true for all GR functions manipulating image data.

FORTRAN

```
integer function sfwdata(sds_id, start, stride, edge, buffer)

integer sds_id
integer start(*), stride(*), edge(*)
<valid numeric data type> buffer(*)

integer function sfwcdata(sds_id, start, stride, edge, buffer)

integer sds_id
integer start(*), stride(*), edge(*)
character*(*) buffer(*)
```
Vaddtagref/vfadtr

int32 Vaddtagref(int32 vgroup_id, int32 tag, int32 ref)

vgroup_id IN: Vgroup identifier returned by Vattach

tag IN: Tag of the object

ref IN: Reference number of the object

Purpose Inserts an object into a vgroup.

Return value Returns the number of objects in the vgroup if successful and FAIL (or -1) otherwise.

Description Vaddtagref inserts the object identified by the parameters tag and ref into the vgroup identified by the parameter vgroup_id.

If an object to be inserted is a data set, duplication of the tag/reference number pair will be allowed. Otherwise, the tag/reference number pair must be unique among the elements within the vgroup or the routine will return FAIL (or -1).

FORTRAN integer function vfadtr(vgroup_id, tag, ref)

 integer vgroup_id, tag, ref
Vattach/vfatch

int32 Vattach(int32 file_id, int32 vgroup_ref, char *access)

file_id IN: File identifier returned by Hopen
vgroup_ref IN: Reference number for the vgroup
access IN: Type of access

Purpose Initiates access to a new or existing vgroup.

Return value Returns the vgroup identifier (vgroup_id) if successful and FAIL (or -1) otherwise.

Description Vattach opens a vgroup with access type specified by the parameter access in the file identified by the parameter file_id. The vgroup is identified by the reference number, vgroup_ref.

Vattach returns the vgroup identifier, vgroup_id, for the accessed vgroup. The vgroup_id is used for all subsequent operations on this vgroup. Once operations are complete, the vgroup identifier must be disposed of via a call to Vdetach. Multiple attaches may be made to the same vgroup simultaneously, and several vgroup identifiers can be created for the same vgroup. Each vgroup identifier must be disposed of independently.

The parameter file_id is the file identifier of an opened file. The parameter vgroup_ref specifies which vgroup in the file to attach to. If vgroup_ref is set to -1, a new vgroup will be created. If vgroup_ref is set to a positive number, the vgroup with that as a reference number is attached.

Possible values for the parameter access are “r” for read access and “w” for write access.

FORTRAN integer function vfatch(file_id, vgroup_ref, access)

integer file_id, vgroup_ref
character*1 access
Vattrinfo/vfainfo

```c
intn Vattrinfo(int32 vgroup_id, intn attr_index, char *attr_name, int32 *data_type, int32 *count, int32 *size)
```

- **vgroup_id**: IN: Vgroup identifier returned by Vattach
- **attr_index**: IN: Index of the attribute
- **attr_name**: OUT: Name of the attribute
- **data_type**: OUT: Data type of the attribute
- **count**: OUT: Number of values in the attribute
- **size**: OUT: Size, in bytes, of the attribute values.

**Purpose**
Retrieves the name, data type, number of values, and value size of an attribute for a vgroup.

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

**Description**
Vattrinfo retrieves the name, datatype, number of values, and value size of an attribute identified by its index, attr_index, in the vgroup, vgroup_id. Name, data type, number of values and size are retrieved into the parameters attr_name, data_type, count, and size, respectively.

If the attribute’s name, data type, number of values, or value size are not needed, the corresponding output parameters can be set to NULL.

The valid value attr_index range from 0 to the total number of attributes attached to a vgroup - 1. The number of vgroup attributes can be obtained using Vnattrs.

**FORTRAN**
```
integer function vfainfo(vgroup_id, attr_index, attr_name, data_type, count, size)

integer vgroup_id, attr_index, data_type, count, size
character*(*) attr_name
```
Vdelete/vdelete

int32 Vdelete(int32 file_id, int32 vgroup_ref)

file_id IN: File identifier returned by Hopen
vgroup_ref IN: Vgroup reference number returned by Vattach

Purpose Remove a vgroup from a file.

Return value Returns SUCCEED (or 0) if successful and FAIL (or -1) if not successful.

Description Vdelete removes the vgroup identified by the parameter vgroup_ref from the
file identified by the parameter file_id.

This routine will remove the vgroup from the internal data structures and from
the file.

FORTRAN

integer function vdelete(file_id, vgroup_ref)

integer file_id, vgroup_ref
int32 Vdeletetagref(int32 vgroup_id, int32 tag, int32 ref)

vgroup_id IN: Vgroup identifier returned by Vattach

tag IN: Tag of the object

ref IN: Reference number of the object

Purpose Deletes an object from a vgroup.

Return value Returns SUCCEED (or 0) if successful and FAIL (or -1) if not successful or the given tag/reference number pair is not found in the vgroup.

Description Vdeletetagref deletes the object specified by the parameters tag and ref from the vgroup identified by the parameter vgroup_id. Vinqtagref should be used to check if the tag/reference number pair exists before calling this routine.

If duplicate tag/reference number pairs are found in the vgroup, Vdeletetagref deletes the first occurrence. Vinqtagref should be used to determine if duplicate tag/reference number pairs exist in the vgroup.

FORTRAN integer function vfdtr(vgroup_id, tag, ref)

integer vgroup_id, tag, ref
**Vdetach/vfdtch**

int32 Vdetach(int32 vgroup_id)

- **vgroup_id**
  - **IN:** Vgroup identifier returned by **Vattach**

- **Purpose**
  - Terminates access to a vgroup.

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

- **Description**
  - **Vdetach** detaches the currently-attached vgroup identified by **vgroup_id** and terminates access to that vgroup.

  All space associated with the vgroup, **vgroup_id**, will be freed. Each attached vgroup must be detached by calling this routine before the file is closed. **Vdetach** also updates the vgroup information in the HDF file if any changes occur. The identifier **vgroup_id** should not be used after the vgroup is detached.

**FORTRAN**

integer function vfdtch(vgroup_id)

integer vgroup_id
Vend/vfend

intn Vend(int32 \textit{file\_id})

\textit{file\_id} \hspace{1em} \text{IN: File identifier returned by Hopen}

\textbf{Purpose} \hspace{1em} Terminates access to a vgroup and/or vdata interface.

\textbf{Return value} \hspace{1em} Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.

\textbf{Description} \hspace{1em} \texttt{Vend} terminates access to the vgroup and/or vdata interfaces initiated by \texttt{Vstart} and all internal data structures allocated by \texttt{Vstart}.

\texttt{Vend} must be called after all vdata and vgroup operations on the file \textit{file\_id} are completed. Further attempts to use vdata or vgroup routines after calling \texttt{Vend} will result in a FAIL (or -1) being returned.

\textbf{FORTRAN} \hspace{1em} integer function vfend(file_id)

integer file_id
Vfind/vfind

int32 Vfind(int32 file_id, char *vgroup_name)

file_id        IN:  File identifier returned by Hopen
vgroup_name    IN:  Name of the vgroup

Purpose  Returns the reference number of a vgroup given its name.
Return value  Returns the reference number of the vgroup if successful and 0 otherwise.
Description  Vfind searches the file identified by the parameter file_id for a vgroup with the
name specified by the parameter vgroup_name, and returns the corresponding
reference number.

If more than one vgroup has the same name, Vfind will return the reference
number of the first one.

FORTRAN

integer function vfind(file_id, vgroup_name)

integer file_id
character*(*) vgroup_name
Vfindattr/vffdatt

intn Vfindattr(int32 vgroup_id, char *attr_name)

vgroup_id    IN: Vgroup identifier returned by Vattach
attr_name    IN: Name of the attribute

Purpose         Returns the index of a vgroup attribute given its name.
Return value    Returns the index of an attribute if successful and FAIL (or -1) otherwise.
Description     Vfindattr searches the vgroup identified by the parameter vgroup_id for the
                 attribute with the name specified by the parameter attr_name, and returns the
                 index of that attribute.
                 If more than one attribute has the same name, Vfindattr will return the index
                 of the first one.

FORTRAN          integer function vffdatt(vgroup_id, attr_name)

               integer vgroup_id
               character*(*) attr_name
## Vfindclass/vfndcls

The function `Vfindclass` searches the file identified by the parameter `file_id` for a vgroup with the class name specified by the parameter `vgroup_class`, and returns the reference number of that vgroup. If more than one vgroup has the same class name, `Vfindclass` will return the reference number of the first one.

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>file_id</code></td>
<td>File identifier returned by <code>Hopen</code></td>
<td>integer</td>
</tr>
<tr>
<td><code>vgroup_class</code></td>
<td>Class name of the vgroup</td>
<td>character</td>
</tr>
</tbody>
</table>

### Purpose

Returns the reference number of a vgroup specified by its class name.

### Return value

Returns the reference number of the vgroup if successful and 0 otherwise.

### Description

**Vfindclass** searches the file identified by the parameter `file_id` for the vgroup with the class name specified by the parameter `vgroup_class`, and returns the reference number of that vgroup.

If more than one vgroup has the same class name, **Vfindclass** will return the reference number of the first one.

### FORTRAN

```fortran
integer function vfndcls(file_id, vgroup_class)

integer file_id
character(*) vgroup_class
```
Vflocate/vffloc

int32 Vflocate(int32 vgroup_id, char *field_name)

vgroup_id IN: Vgroup identifier returned by Vattach
field_name_list IN: List of field names

Purpose Locates a vdata in a vgroup given a list of field names.

Return value Returns the reference number of the vdata if successful and FAIL (or -1) otherwise.

Description Vflocate searches the vgroup identified by the parameter vgroup_id for a vdata that contains all of the fields listed in the parameter field_name_list. If that vdata is found, Vflocate will return its reference number.

FORTRAN integer function vffloc(vgroup_id, field_name)

integer vgroup_id
character*(*) field_name
Vgetattr/vfgnatt/vfgcatt

Vgetattr/vfgnatt/vfgcatt

intn Vgetattr(int32 vgroup_id, intn attr_index, VOIDP attr_values)

vgroup_id        IN:     Vgroup identifier returned by Vattach
attr_index       IN:     Index of the attribute
attr_values      OUT:    Buffer for the attribute values

Purpose         Retrieves the values of a vgroup attribute.
Return value     Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description     Vgetattr retrieves the values of the attribute identified by its index, attr_index,
                 into the buffer attr_values for the vgroup identified by the parameter vgroup_id.

The valid values of the parameter attr_index range from 0 to the total number of
vgroup attributes - 1. The total number of attributes can be obtained using
Vnattrs. To determine the amount of memory sufficient to hold the attribute
values, the user can obtain the number of attribute values and the attribute
value size using Vattrinfo.

FORTRAN

integer function vfgnatt(vgroup_id, attr_index, attr_values)
integer vgroup_id, attr_index
<valid numeric data type> attr_values

integer function vfgcatt(vgroup_id, attr_index, attr_values)
integer vgroup_id, attr_index
character*(*) attr_values
Vgetclass/vfgcls

int32 Vgetclass(int32 vgroup_id, char *vgroup_class)

vgroup_id        IN:   Vgroup identifier returned by Vattach
vgroup_class     OUT:   Class name of the vgroup

Purpose          Retrieves the class name of a vgroup.
Return value     Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description      Vgetclass retrieves the class name of the vgroup identified by the parameter vgroup_id in the buffer vgroup_class.

FORTRAN
integer function vfgcls(vgroup_id, vgroup_class)

integer vgroup_id
character*(*) vgroup_class
Vgetid/vfgid

Vgetid/vfgid

int32 Vgetid(int32 file_id, int32 vgroup_ref)

file_id IN: File identifier returned by Hopen
vgroup_ref IN: Reference number of the current vgroup

Purpose
Returns the reference number of the next vgroup.

Return value
Returns the reference number of the next vgroup if successful and FAIL (or -1) otherwise.

Description
Vgetid sequentially searches the file identified by the parameter file_id and returns the reference number of the vgroup following the vgroup that has the reference number specified by the parameter vgroup_ref.

The search is initiated by calling this routine with a vgroup_ref value of -1. This will return the reference number of the first vgroup in the file. Searching past the last vgroup in the file will cause Vgetid to return FAIL (or -1).

FORTRAN
integer function vfgid(file_id, vgroup_ref)

integer file_id, vgroup_ref
Vgetname/vfgnam

int32 Vgetname(int32 vgroup_id, char *vgroup_name)

vgroup_id IN: Vgroup identifier returned by Vattach
vgroup_name OUT: Name of the vgroup

Purpose Retrieves the name of a vgroup.
Return value Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description Vgetname retrieves the name of the vgroup identified by the parameter vgroup_id into the buffer vgroup_name. The maximum length of the name is defined by VGNAMELENMAX (or 64).

FORTRAN integer function vfgnam(vgroup_id, vgroup_name)

integer vgroup_id
character*(*) vgroup_name
Vgetnext/vfgnxt

int32 Vgetnext(int32 vgroup_id, int32 v_ref)

vgroup_id    IN:  Vgroup identifier returned by Vattach
v_ref        IN:  Reference number of the vgroup or vdata

Purpose       Gets the reference number of the next member (vgroup or vdata only) of a vgroup.

Return value  Returns the reference number of the vgroup or vdata if successful and FAIL (or -1) otherwise.

Description   Vgetnext searches in the vgroup identified by the parameter vgroup_id for the object following the object specified by its reference number v_ref. Either of the two objects can be a vgroup or a vdata. If v_ref is set to -1, the routine will return the reference number of the first vgroup or vdata in the vgroup.

Note that this routine only gets a vgroup or a vdata in a vgroup. Vgettagrefs gets any object in a vgroup.

FORTRAN       integer function vfgnxt(vgroup_id, v_ref)

integer vgroup_id, v_ref

National Center for Supercomputing Applications
Vgettagref/vfgttr

```
intn Vgettagref(int32 vgroup_id, int32 index, int32 *tag, int32 *ref)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vgroup_id</td>
<td>Vgroup identifier returned by Vattach</td>
</tr>
<tr>
<td>index</td>
<td>Index of the object in the vgroup</td>
</tr>
<tr>
<td>tag</td>
<td>Tag of the object</td>
</tr>
<tr>
<td>ref</td>
<td>Reference number of the object</td>
</tr>
</tbody>
</table>

**Purpose**

Retrieves the tag/reference number pair of an object given its index within a vgroup.

**Return value**

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

**Description**

Vgettagref retrieves the tag/reference number pair of the object specified by its index, `index`, within the vgroup identified by the parameter `vgroup_id`. Note that this routine is different from Vgettagrefs, which retrieves the tag/reference number pairs of a number of objects.

The valid values of `index` range from 0 to the total number of objects in the vgroup - 1. The total number of objects in the vgroup can be obtained using Vinquire.

The tag is stored in the buffer `tag` and the reference number is stored in the buffer `ref`.

**FORTRAN**

```
integer function vfgttr(vgroup_id, index, tag, ref)
```

```fortran
integer vgroup_id, index
integer tag, ref
```
Vgettagrefs/vfgttrs

int32 Vgettagrefs(int32 vgroup_id, int32 tag_array[], int32 ref_array[], int32 num_of_pairs)

vgroup_id IN: Vgroup identifier returned by Vattach
tag_array OUT: Array of tags
ref_array OUT: Array of reference numbers
num_of_pairs IN: Number of tag/reference number pairs

Purpose
Retrieves the tag/reference number pairs of the HDF objects belonging to a vgroup.

Return value
Returns the number of tag/reference number pairs obtained from a vgroup if successful and \texttt{FAIL} (or -1) otherwise.

Description
Vgettagrefs retrieves at most \texttt{num_of_pairs} number of tag/reference number pairs belonging to the vgroup, \texttt{vgroup_id}, and stores them in the buffers \texttt{tag_array} and \texttt{ref_array}.

The input parameter \texttt{num_of_pairs} specifies the maximum number of tag/reference number pairs to be returned. The size of the arrays, \texttt{tag_array} and \texttt{ref_array}, must be at least \texttt{num_of_pairs}.

FORTRAN

\begin{verbatim}
integer function vfgttrs(vgroup_id, tag_array, ref_array, num_of_pairs)
  integer vgroup_id, num_of_pairs
  integer tag_array(*), ref_array(*)
\end{verbatim}
Vgetversion/vfgver

int32 Vgetversion(int32 vgroup_id)

vgroup_id   IN: Vgroup identifier returned by Vattach

Purpose Gets the version of a vgroup.

Return value Returns the vgroup version number if successful, and FAIL (or -1) otherwise.

Description Vgetversion returns the version number of the vgroup identified by the parameter vgroup_id. There are three valid version numbers: VSET_OLD_VERSION (or 2), VSET_VERSION (or 3), and VSET_NEW_VERSION (or 4).

VSET_OLD_VERSION is returned when the vgroup is of a version that corresponds to an HDF library version before version 3.2.

VSET_VERSION is returned when the vgroup is of a version that corresponds to an HDF library version between versions 3.2 and 4.0 release 2.

VSET_NEW_VERSION is returned when the vgroup is of the version that corresponds to an HDF library version of version 4.1 release 1 or higher.

FORTRAN integer function vfgver(vgroup_id)

integer vgroup_id
Vinqtagref/vfinqtr

intn Vinqtagref(int32 vgroup_id, int32 tag, int32 ref)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vgroup_id</td>
<td>Vgroup identifier returned by Vattach</td>
</tr>
<tr>
<td>tag</td>
<td>Tag of the object</td>
</tr>
<tr>
<td>ref</td>
<td>Reference number of the object</td>
</tr>
</tbody>
</table>

**Purpose**
Checks whether an object belongs to a vgroup.

**Return value**
Returns TRUE (or 1) if the object belongs to the vgroup, and FALSE (or 0) otherwise.

**Description**
Vinqtagref checks if the object identified by its tag, tag, and its reference number, ref, belongs to the vgroup identified by the parameter vgroup_id.

**FORTRAN**

```fortran
integer function vfinqtr(vgroup_id, tag, ref)

integer vgroup_id, tag, ref
```
**Vinquire/vfinq**

```c
intn Vinquire(int32 vgroup_id, int32 *n_entries, char *vgroup_name)
```

- `vgroup_id`: IN: Vgroup identifier returned by `Vattach`
- `n_entries`: OUT: Number of entries in a vgroup
- `vgroup_name`: OUT: Name of a vgroup

**Purpose**
Retrieves the number of entries in a vgroup and its name.

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

**Description**
`Vinquire` retrieves the name of and the number of entries in the vgroup identified by the parameter `vgroup_id` into the buffer `vgroup_name` and the parameter `n_entries`, respectively.

The maximum length of the vgroup name is defined by `VGNAMELENMAX` (or 64).

**FORTRAN**

```fortran
integer function vfinq(vgroup_id, n_entries, vgroup_name)

integer vgroup_id, n_entries
character(*) vgroup_name
```
Vinser\textbackslash{}vfinsrt

\texttt{int32 Vinser(int32 \textit{vgroup\_id}, int32 \textit{v\_id})}

\begin{itemize}
\item \texttt{vgroup\_id} \quad \textbf{IN:} \quad \text{Vgroup identifier returned by} \texttt{Vattach}
\item \texttt{v\_id} \quad \textbf{IN:} \quad \text{Identifier of the vdata or vgroup}
\end{itemize}

\textbf{Purpose} \quad \text{Inserts a vdata or vgroup into a vgroup.}

\textbf{Return value} \quad \text{Returns the position (\textit{index}) of the inserted element within the vgroup if successful and \texttt{FAIL} (or \texttt{-1}) otherwise.}

\textbf{Description} \quad \texttt{Vinser} inserts the vdata or vgroup identified by the parameter \texttt{v\_id} into the vgroup identified by the parameter \texttt{vgroup\_id}.

\text{Essentially,} \texttt{Vinser} \text{only inserts a vgroup or vdata. To insert any objects into a vgroup, use} \texttt{Vaddtagref}.

\text{The returned value,} \textit{index}, \text{is either 0 or a positive value, which indicates the position of the inserted element in the vgroup.}

\textbf{FORTRAN} \quad \texttt{integer function vfinsrt(vgroup\_id, v\_id)}

\texttt{integer vgroup\_id, v\_id}


**Visvg/vfisvg**

```c
intn Visvg(int32 vgroup_id, int32 obj_ref)
```

**vgroup_id**  
IN:  Vgroup identifier returned by `Vattach`

**obj_ref**  
IN:  Reference number of the object

**Purpose**  
Determines whether an element of a vgroup is a vgroup and a member of another vgroup.

**Return value**  
Returns `TRUE` (or 1) if the object is a vgroup and `FALSE` (or 0) otherwise.

**Description**  
`Visvg` determines if the object specified by the reference number, `obj_ref`, is a vgroup within the vgroup identified by the parameter `vgroup_id`.

**FORTRAN**

```fortran
integer function vfisvg(vgroup_id, obj_ref)
```

```fortran
integer vgroup_id, obj_ref
```
### Visvs/vfisvs

**Visvs/vfisvs**

```fortran
intn Visvs(int32 vgroup_id, int32 obj_ref)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vgroup_id</code></td>
<td>Vgroup identifier returned by <strong>Vattach</strong></td>
</tr>
<tr>
<td><code>obj_ref</code></td>
<td>Reference number of the object</td>
</tr>
</tbody>
</table>

**Purpose**

Determines whether a data object is a vdata within a vgroup.

**Return value**

Returns **TRUE** (or 1) if the object is a vdata and **FALSE** (or 0) otherwise.

**Description**

**Visvs** determines if the object specified by the reference number, `obj_ref`, is a vdata within the vgroup identified by the parameter `vgroup_id`.

**FORTRAN**

```fortran
integer function vfisvs(vgroup_id, obj_ref)
integer vgroup_id, obj_ref
```
Vlone/vflone

int32 Vlone(int32 file_id, int32 ref_array[], int32 max.refs)

file_id IN: File identifier returned by Hopen

ref_array OUT: Array of reference numbers

max.refs IN: Maximum number of lone vgroups to be retrieved

Purpose Retrieves the reference numbers of lone vgroups, i.e., vgroups that are at the
top of the grouping hierarchy, in a file.

Return value Returns the total number of lone vgroups if successful and FAIL (or −1)
otherwise.

Description Vlone retrieves the reference numbers of lone vgroups in the file identified by
the parameter file_id. Although Vlone returns the total number of lone
vgroups in the file, only at most max.refs reference numbers are retrieved and
stored in the buffer ref_array. The array must have at least max.refs elements.

An array size of 65,000 integers for ref_array is more than adequate if the user
chooses to declare the array statically. However, the preferred method is to
dynamically allocate memory instead; first call Vlone with a value of 0 for
max.refs, and then use the returned value to allocate memory for ref_array
before calling Vlone again.

FORTRAN integer function vflone(file_id, ref_array, max.refs)

integer file_id, ref_array(*), max.refs
Vnattrs/vfnatts

Vnattrs/vfnatts

int n Vnattrs(int32 vgroup_id)

vgroup_id IN: Vgroup identifier returned by Vattach

Purpose

Returns the number of attributes assigned to a vgroup.

Return value

Returns the total number of attributes assigned to the specified vgroups if successful and FAIL (or -1) otherwise.

Description

Vnattrs gets the number of attributes assigned to the vgroup identified by the parameter vgroup_id.

FORTRAN

integer function vfnatts(vgroup_id)

integer vgroup_id
Vnrefs/vnrefs

int32 Vnrefs(int32 vgroup_id, int32 tag_type)

vgroup_id IN: Vgroup identifier returned by Vattach
tag_type IN: Type of the tag

Purpose Returns the number of tags of a given tag type in a vgroup.
Return value Returns 0 or the total number of tags if successful and FAIL (or -1) otherwise.
Description Vnrefs returns 0 or the number of tags having the type specified by the parameter tag_type in the vgroup identified by the parameter vgroup_id.

See Appendix A, NCSA HDF Tags, in the HDF User’s Guide, for a discussion of tag types.

FORTRAN integer function vnrefs(vgroup_id, tag_type)

integer vgroup_id, tag_type
**Vntagrefs/vfntr**

```
int32 Vntagrefs(int32 vgroup_id)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vgroup_id</td>
<td>IN: Vgroup identifier returned by <strong>Vattach</strong></td>
</tr>
</tbody>
</table>

**Purpose**

Returns the number of objects in a vgroup.

**Return value**

Returns 0 or a positive number representing the number of HDF objects linked to the vgroup if successful or **FAIL** (or -1) otherwise.

**Description**

**Vntagrefs** returns the number of objects in a vgroup identified by the parameter `vgroup_id`.

**Vntagrefs** is used together with **Vgettagrefs**, or with **Vgettagref** to look at the data objects linked to a given vgroup.

**FORTRAN**

```
integer function vfntr(vgroup_id)

integer vgroup_id
```
Vsetattr/vfsnatt/vfscatt

intn Vsetattr(int32 vgroup_id, char *attr_name, int32 data_type, int32 count, VOIDP values)

vgroup_id  IN: Vgroup identifier returned by Vattach
attr_name  IN: Name of the attribute
data_type  IN: Data type of the attribute
count  IN: Number of values the attribute contains
values  IN: Buffer containing the attribute values

Purpose
Attaches an attribute to a vgroup.

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

Description
Vsetattr attaches an attribute to the vgroup identified by the parameter vgroup_id. The attribute name is specified by the parameter attr_name and the attribute data type is specified by the parameter data_type. The values of the attribute are specified by the parameter values, and the number of values in the attribute is specified by the parameter count. Refer to Table 1A in Section I of this manual for a listing of all valid data types.

If the attribute already exists, the new values will replace the current ones, provided the data type and the number of attribute values have not been changed. If either the data type or the order have been changed, Vsetattr will return FAIL (or -1).

FORTRAN
integer vfsnatt(vgroup_id, attr_name, data_type, count, values)
integer vgroup_id, data_type, count
<valid numeric data type> values(*)
character*(*) attr_name

integer vfscatt(vgroup_id, attr_name, data_type, count, values)
integer vgroup_id, data_type, count
character*(*) attr_name, values(*)
Vsetclass/vfscls

Vsetclass/vfscls

int32 Vsetclass(int32 vgroup_id, char *vgroup_class)

vgroup_id       IN: Vgroup identifier returned by Vattach
vgroup_class    IN: Class name of a vgroup

Purpose       Sets the class name of a vgroup.
Return value  Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description   Vsetclass sets the class name specified by the parameter vgroup_class to the vgroup identified by the parameter vgroup_id.

A vgroup initially has a class name of NULL. The class name may be set more than once. Class names, like vgroup names, can be of any character strings. They exist solely as meaningful labels for user applications.

The class name is limited to VSNAMELENMAX (or 64) characters.

FORTRAN  

terger function vfscls(vgroup_id, vgroup_class)

integer vgroup_id
character(*) vgroup_class
Vsetname/vfsnam

```c
int32 Vsetname(int32 vgroup_id, char *vgroup_name)
```

- **vgroup_id** (IN): Vgroup identifier returned by `Vattach`
- **vgroup_name** (IN): Name of a vgroup

**Purpose**
Sets the name of a vgroup.

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

**Description**
`Vsetname` sets the name specified by the parameter `vgroup_name` for the vgroup identified by the parameter `vgroup_id`.

A vgroup initially has a name of `NULL`, and may be renamed more than once during the scope of the vgroup identifier (`vgroup_id`). Note that the routine does not check for uniqueness of vgroup names.

Vgroup names are optional, but recommended. They serve as meaningful labels for user applications. If used, they should be unique. The name length is limited to `VSNAMELENMAX` (or 64) characters.

**FORTRAN**

```fortran
integer function vfsnam(vgroup_id, vgroup_name)

integer vgroup_id
character(*) vgroup_name
```
Vstart/vfstart

int Vstart(int32 file_id)

file_id IN: File identifier returned by Hopen

Purpose Initializes the vdata and/or vgroup interface.

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

Description Vstart initializes the vdata and/or vgroup interfaces for the file identified by the parameter file_id.

Vstart must be called before any vdata or vgroup operation is attempted on an HDF file. Vstart must be called once for each file involved in the operation.

FORTRAN integer function vfstart(file_id)

integer file_id
VHmakegroup/vhfmkgp

VHmakegroup/vhfmkgp

\texttt{int32 \textbf{VHmakegroup}(int32 \texttt{file\_id}, int32 \texttt{tag\_array}[], int32 \texttt{ref\_array}[], int32 \texttt{n\_objects}, char *\texttt{vgroup\_name}, char *\texttt{vgroup\_class})}

\begin{itemize}
\item \texttt{file\_id} \quad \textbf{IN}: File identifier returned by \texttt{Hopen}
\item \texttt{tag\_array} \quad \textbf{IN}: Array of tags
\item \texttt{ref\_array} \quad \textbf{IN}: Array of reference numbers
\item \texttt{n\_objects} \quad \textbf{IN}: Number of data objects to be stored
\item \texttt{vgroup\_name} \quad \textbf{IN}: Name of the vgroup
\item \texttt{vgroup\_class} \quad \textbf{IN}: Class of the vgroup
\end{itemize}

\textbf{Purpose} Creates a vgroup.

\textbf{Return value} Returns the reference number of the newly-created vgroup if successful, \texttt{FAIL} (or -1) otherwise.

\textbf{Description} \texttt{VHmakegroup} creates a vgroup with the name specified by the parameter \texttt{vgroup\_name} and the class name specified by the parameter \texttt{vgroup\_class} in the file identified by the parameter \texttt{file\_id}. The routine inserts \texttt{n\_objects} objects into the vgroup. The tag and reference numbers of the objects to be inserted are specified in the arrays \texttt{tag\_array} and \texttt{ref\_array}.

Creating empty vgroups with \texttt{VHmakegroup} is allowed. \texttt{VHmakegroup} does not check if the tag/reference number pair is valid, or if the corresponding data object exists. However, all of the tag/reference number pairs must be unique.

\texttt{Vstart} must precede any calls to \texttt{VHmakegroup}. It is not necessary, however, to call \texttt{Vattach} or \texttt{Vdetach} in conjunction with \texttt{VHmakegroup}.

The elements in the arrays \texttt{tag\_array} and \texttt{ref\_array} are the matching tag/reference number pairs of the objects to be inserted, that means \texttt{tag\_array[0]} and \texttt{ref\_array[0]} refer to one data object, and \texttt{tag\_array[1]} and \texttt{ref\_array[1]} to another, etc.

\textbf{FORTRAN} integer function vhfmkgp(file_id, tag_array, ref_array, n_objects, vgroup_name, vgroup_class)

\begin{verbatim}
integer file_id, n_objects
character(*) vgroup_name, vgroup_class
integer tag_array(*), ref_array(*)
\end{verbatim}
VQueryref/vqref

int32 VQueryref(int32 vgroup_id)

vgroup_id    IN:       Vgroup identifier returned by Vattach

Purpose        Returns the reference number of a vgroup.
Return value   Returns the reference number if successful, and FAIL (or -1) otherwise.
Description    VQueryref returns the reference number of the vgroup identified by the parameter vgroup_id.

FORTRAN          integer function vqref(vgroup_id)

               integer vgroup_id
VQuerytag/vqtag

int32 VQuerytag(int32 vgroup_id)

vgroup_id IN: Vgroup identifier returned by Vattach

Purpose Returns the tag of a vgroup.
Return value Returns the tag if successful, and FAIL (or -1) otherwise.
Description VQuerytag returns the tag of the vgroup identified by the parameter vgroup_id.

FORTRAN integer function vqtag(vgroup_id)

integer vgroup_id
VFfieldesize/vffesiz

int32 VFfieldesize(int32 vdata_id, int32 field_index)

vdata_id IN: Vdata identifier returned by VSattach
field_index IN: Vdata field index

Purpose Returns the size, as stored on disk, of a vdata field.

Return value Returns the vdata field size if successful and FAIL (or -1) otherwise.

Description VFfieldesize returns the size, as stored on disk, of a vdata field identified by
the parameter field_index in the vdata identified by the parameter vdata_id.

The value of the parameter field_index ranges from 0 to the total number of
fields in the vdata - 1. The number of vdata fields is returned by VFfields
function.

FORTRAN integer function vffesiz(vdata_id, field_index)

integer vdata_id, field_index
VFfieldisize/vffisiz

VFfieldisize/vffisiz

```c
int32 VFfieldisize(int32 vdata_id, int32 field_index)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vdata_id</td>
<td>Vdata identifier returned by VSattach</td>
</tr>
<tr>
<td>field_index</td>
<td>Vdata field index</td>
</tr>
</tbody>
</table>

**Purpose**

Returns the size, as stored in memory, of a vdata field.

**Return value**

Returns the vdata field size if successful and FAIL (or -1) otherwise.

**Description**

VFfieldisize returns the size, as stored in memory, of a vdata field identified by the parameter field_index in the vdata identified by the parameter vdata_id.

The value of the parameter field_index ranges from 0 to the total number of fields in the vdata - 1. The number of vdata fields is returned by VFfields function.

**FORTRAN**

```fortran
integer function vffisiz(vdata_id, field_index)
```

```fortran
integer vdata_id, field_index
```
VFfieldname/vffname

char *VFfieldname(int32_vdata_id, int32_field_index)

vdata_id IN: Vdata identifier returned by VSattach
field_index IN: Vdata field index

Purpose
Returns the name of a vdata field.

Return value
Returns a pointer to the vdata field name if successful and NULL otherwise. The FORTRAN-77 version of this routine, vffname, returns SUCCEED (or 0) or FAIL (or -1).

Description
VFfieldname returns the name of the vdata field identified by the parameter field_index in the vdata identified by the parameter vdata_id.

The value of the parameter field_index ranges from 0 to the total number of fields in the vdata - 1. The number of vdata fields is returned by VFfields function.

The FORTRAN-77 version of this routine, vffname, returns the field name in the parameter fname.

FORTRAN
integer function vffname(vdata_id, field_index, fname)

integer vdata_id, field_index
character*(*) fname
VFfieldorder/vffordr

VFfieldorder/vffordr

int32 VFfieldorder(int32 vdata_id, int32 field_index)

vdata_id     IN:  Vdata identifier returned by VSattach
field_index  IN:  Vdata field index

Purpose       Returns the order of a vdata field.
Return value  Returns the order of the field if successful and FAIL (or -1) otherwise.
Description   VFfieldorder returns the order of the vdata field identified by its index, field_index, in the vdata identified by the parameter vdata_id.

The value of the parameter field_index ranges from 0 to the total number of fields in the vdata - 1. The number of vdata fields is returned by VFfields function.

FORTRAN      integer function vffordr(vdata_id, field_index)

integer vdata_id, field_index
**VFfieldtype/vfftype**

\[
\text{int32 VFfieldtype(int32 } vdata\_id, \text{ int32 } field\_index)\]

*Purpose*  
Returns the data type of a vdata field.

*Return value*  
Returns the data type if successful and FAIL (or -1) otherwise.

*Description*  
\text{VFfieldtype} returns the data type of the vdata field identified by its index, \textit{field\_index}, in the vdata identified by the parameter \textit{vdata\_id}.  
The value of the parameter \textit{field\_index} ranges from 0 to the total number of fields in the vdata - 1. The number of vdata fields is returned by \text{VFnfields} function.

*FORTRAN*  
\[
\text{integer function vfftype(vdata\_id, field\_index)}
\]

\[
\text{integer vdata\_id, field\_index}
\]
VFnfields/vfnflds

VFnfields/vfnflds

int32 VFnfields(int32 vdata_id)

<table>
<thead>
<tr>
<th>vdata_id</th>
<th>IN: Vdata identifier returned by VSatattach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Returns the total number of fields in a vdata.</td>
</tr>
<tr>
<td>Return value</td>
<td>Returns the total number of fields if successful and FAIL (or -1) otherwise.</td>
</tr>
<tr>
<td>Description</td>
<td>VFnfields returns the total number of fields in the vdata identified by the parameter vdata_id.</td>
</tr>
</tbody>
</table>

FORTRAN

integer function vfnflds(vdata_id)

integer vdata_id
VSQuerycount/vsqfnelt

```c
intn VSQuerycount(int32 vdata_id, int32 *n_records)
```

- **vdata_id**
  - **IN:** Vdata access identifier returned by VSAttach
- **n_records**
  - **OUT:** Number of records in the vdata

**Purpose**
Retrieves the number of records in a vdata.

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

**Description**
VSQuerycount retrieves the number of records in the vdata identified by `vdata_id` in the parameter `n_records`.

**FORTRAN**
```fortran
integer function vsqfnelt(vdata_id, n_records)

integer vdata_id, n_records
```
### VSQueryfields/vsqfflds

`intn VSQueryfields(int32 vdata_id, char *field_name_list)`

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vdata_id</code></td>
<td><code>int32</code></td>
<td>Vdata access identifier returned by <code>VSattach</code></td>
</tr>
<tr>
<td><code>field_name_list</code></td>
<td><code>char *</code></td>
<td>List of field names</td>
</tr>
</tbody>
</table>

#### Purpose
Retrieves the names of the fields in a vdata.

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

#### Description
`VSQueryfields` retrieves the names of the fields in the vdata identified by the parameter `vdata_id` into the parameter `field_name_list`.

The parameter `field_name_list` is a comma-separated list of the fields in the vdata. (i.e., “PX, PY, PZ” in C and ’PX, PY, PZ’ in Fortran).

#### FORTRAN

```
integer function vsqfflds(vdata_id, field_name_list)

integer vdata_id

character*(*) field_name_list
```
VSQueryinterlace/vsqfintr

```c
intn VSQueryinterlace(int32 vdata_id, int32 *interlace_mode)
```

- **vdata_id** IN: Vdata identifier returned by VSAttach
- **interlace_mode** OUT: Interlace mode

**Purpose**
Retrieves the interlace mode of the vdata.

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

**Description**
VSQueryinterlace retrieves the interlace mode of the vdata identified by the parameter `vdata_id` into the parameter `interlace_mode`.

Valid values for `interlace_mode` are FULL_INTERLACE (or 0) and NO_INTERLACE (or 1).

**FORTRAN**
```fortran
integer function vsqfintr(vdata_id, interlace_mode)
```

```fortran
integer vdata_id, interlace_mode
```
VSQueryname/vsqfname

intn VSQueryname(int32 vdata_id, char *vdata_name)

vdata_id IN: Vdata identifier returned by VSattach
vdata_name OUT: Name of the vdata

Purpose Retrieves the name of a vdata.
Return value Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description VSQueryname retrieves the name of the vdata identified by the parameter vdata_id into the buffer vdata_name.
The buffer vdata_name should be set to at least VSNAMELENMAX bytes. VSNAMELENMAX is defined by the HDF library.

FORTRAN integer function vsqfname(vdata_id, vdata_name)

integer vdata_id

character*(*) vdata_name
VSQueryref/vsqref

int32 VSQueryref(int32 vdata_id)

vdata_id  IN:  Vdata identifier returned by VSAttach

Purpose  Returns the reference number of a vdata.

Return value  Returns the reference number of the vdata if successful and FAIL (or -1) otherwise.

Description  VSQueryref returns the reference number of the vdata identified by the parameter vdata_id.

FORTRAN  

integer function vsqref(vdata_id)

integer vdata_id
VSQueryTag/vsqtag

int32 VSQueryTag(int32 vdata_id)

vdata_id IN: Vdata identifier returned by VSAttach

Purpose Returns the tag of the specified vdata.

Return value Returns the tag of the vdata if successful and FAIL (or -1) otherwise.

Description Returns the tag of the vdata identified by the parameter vdata_id.

FORTRAN integer function vsqtag(vdata_id)

integer vdata_id
VSQueryvsize/vsqfvsiz

intn VSQueryvsize(int32 *vdata_id, int32 *vdata_size)

vdata_id IN: Vdata identifier returned by VSAttach
vdata_size OUT: Size of the vdata record

Purpose Retrieves the size of a record in a vdata.
Return value Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description VSQueryvsize retrieves the size, in bytes, of a record in the vdata identified by the parameter vdata_id into the parameter vdata_size. The returned size value is machine dependent.

FORTRAN integer function vsqfvsiz(vdata_id, vdata_size)

integer vdata_id, vdata_size
VHstoredata/vhfsd/vhfscd

int32 VHstoredata(int32 file_id, char *fieldname, uint8 buf[], int32 n_records, int32 data_type, char *vdata_name, char *vdata_class)

- **file_id** IN: File identifier returned by Hopen
- **fieldname** IN: Field name for the new vdata
- **buf** IN: Buffer containing the records to be stored
- **n_records** IN: Number of records to be stored
- **data_type** IN: Type of data to be stored
- **vdata_name** IN: Name of the vdata to be created
- **vdata_class** IN: Class of the vdata to be created

**Purpose**
Creates and writes to a single-field vdata.

**Return value**
Returns reference number of the newly-created vdata if successful, and FAIL (or -1) otherwise.

**Description**
VHstoredata creates a single-field vdata in the file, file_id, and stores data from the buffer buf in it. Vdata name, class name and data type are specified by the parameters vdata_name, vdata_class, and data_type, respectively. Number of records in a vdata is specified by the parameter n_records. Field name is specified by the parameter fieldname.

Vstart must precede VHstoredata. It is not necessary, however, to call VSattach or VSdetach in conjunction with VHstoredata.

This routine provides a high-level method for creating single-order, single-field vdatas.

Note that there are two FORTRAN-77 versions of this routine; one for numeric data (vhfsd) and the other for character data (vhfsdc).

**FORTRAN**

```fortran
integer function vhfsd(file_id, fieldname, buf, n_records, data_type, vdata_name, vdata_class)

integer file_id, n_records, data_type
character(*) vdata_name, vdata_class, fieldname
<valid numeric data type> buf(*)

integer function vhfscd(file_id, fieldname, buf, n_records, data_type, vdata_name, vdata_class)

integer file_id, n_records, data_type
```

December 30, 2003 2-191
character*(*) vdata_name, vdata_class, fieldname
character*(*) buf
VHstoredatam/vhfsdm/vhfscdm

int32 VHstoredatam(int32 file_id, char *fieldname, uint8 buf[], int32 n_records, int32 data_type, char *vdata_name, char *vdata_class, int32 order)

file_id      IN:  File identifier returned by Hopen
fieldname    IN:  Field name
buf          IN:  Buffer containing the records to be stored
n_records    IN:  Number of records to be stored
data_type    IN:  Type of data to be stored
vdata_name   IN:  Name of the vdata to be created
vdata_class  IN:  Class of the vdata to be created
order        IN:  Field order

Purpose
Creates and writes to a multi-order, single-field vdata.

Return value
Returns the reference number of the newly created vdata if successful, and FAIL (or -1) otherwise.

Description
VHstoredatam creates a vdata with the name specified by the parameter vdata_name and a class name specified by the parameter vdata_class in the file identified by the parameter file_id. The data type of the vdata is specified by the parameter data_type. The vdata contains one field with the name specified by the parameter fieldname. The order of the field, order, indicates the number of vdata values stored per field. The vdata contains the number of records specified by the parameter n_records. The buf parameter should contain n_records records that will be stored in the vdata.

Vstart must precede VHstoredatam. It is not necessary, however, to call VStart or VStop in conjunction with VHstoredatam.

This routine provides a high-level method for creating multi-order, single-field vdatas.

Note that there are two FORTRAN-77 versions of this routine; one for numeric data (vhfsdm) and the other for character data (vhfscdm).

FORTRAN

integer function vhfsdm(file_id, fieldname, buf, n_records,
integer file_id, n_records, data_type, order
character(*) vdata_name, vdata_class, fieldname
<valid numeric data type> buf(*))
integer function vhfscdm(file_id, fieldname, buf, n_records, data_type, vdata_name, vdata_class, order)

integer file_id, n_records, data_type, order
character(*) vdata_name, vdata_class, fieldname
character(*) buf
VSappendable/vsapp (Obsolete)

int32 VSappendable(int32 vdata_id, int32 block_size)

vdata_id  IN:  Vdata identifier returned by VSattach
block_size IN:  Standard block size of appended data

Purpose  Makes it possible to append to a vdata.
Return value  Retrieves SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description  The HDF library makes all vdatas appendable upon creation. Therefore, this routine has been made obsolete.

FORTRAN  integer function vsapp(vdata_id, block_size)

integer vdata_id, block_size
**VSattach/vsfatch**

`int32 VSattach(int32 file_id, int32 vdata_ref, char *access)`

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>file_id</code></td>
<td>IN: File identifier returned by <strong>Hopen</strong></td>
</tr>
<tr>
<td><code>vdata_ref</code></td>
<td>IN: Reference number of the vdata</td>
</tr>
<tr>
<td><code>access</code></td>
<td>IN: Access mode</td>
</tr>
</tbody>
</table>

**Purpose**
Attaches to an existing vdata or creates a new vdata.

**Return value**
Returns a vdata identifier if successful and **FAIL** (or -1) otherwise.

**Description**
**VSattach** attaches to the vdata identified by the reference number, `vdata_ref`, in the file identified by the parameter `file_id`. Access to the vdata is specified by the parameter `access`. **VSattach** returns an identifier to the vdata, through which all further operations on that vdata are carried out.

An existing vdata may be multiply-attached for reads. Only one attach with write access to a vdata is allowed.

The default interlace mode for a new vdata is **FULL_INTERLACE** (or 0). This may be changed using **VSetInterlace**.

The value of the parameter `vdata_ref` may be -1. This is used to create a new vdata.

Valid values for `access` are “r” for read access and “w” for write access.

If `access` is “r”, then `vdata_ref` must be the valid reference number of an existing vdata returned from any of the vdata and vgroup search routines (e.g., **Vgetnext** or **VGetId**). It is an error to attach to a vdata with a `vdata_ref` of -1 with “r” access.

If `access` is “w”, then `vdata_ref` must be the valid reference number of an existing vdata or -1. An existing vdata is generally attached with “w” access to replace part of its data, or to append new data to it.

**FORTRAN**

```fortran
integer function vsfatch(file_id, vdata_ref, access)
integer file_id, vdata_ref
character*1 access
```
VSattrinfo/vsfainf

intn VSattrinfo(int32 vdata_id, int32 field_index, intn attr_index, char *attr_name, intn *data_type, intn *count, intn *size)

vdata_id IN: Vdata identifier returned by VSattach
field_index IN: Index of the field
attr_index IN: Index of the attribute
attr_name OUT: Name of the attribute
data_type OUT: Data type of the attribute
count OUT: Attribute value count
size OUT: Size of the attribute

Purpose
Retrieves attribute information of a vdata or a vdata field.

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

Description
VSattrinfo gets information on the attribute attached to the vdata, vdata_id, or to the vdata field. Vdata field is specified by its index, field_index. Attribute is specified by its index, attr_index. The attribute name is returned into the parameter attr_name, the data type is returned into the parameter data_type, the number of values of the attribute is returned into the parameter count, and the size of the attribute is returned into the parameter size.

The parameter field_index in VSattrinfo is the same as the parameter field_index in VSsetattr. It can be set to either an integer field index for the vdata field attribute, or _HDF_VDATA (or -1) to specify the vdata attribute.

In C the values of the parameters attr_name, data_type, count and size can be set to NULL if the information returned by these parameters is not needed.

FORTRAN
integer function vsfainf(vdata_id, field_index, attr_index, attr_name, data_type, count, size)

integer vdata_id, field_index, attr_index
character*(*) attr_name
integer data_type, count, size
## VSdelete/vsfdlte

The function VSdelete removes a vdata from a file. It takes two parameters:

- `file_id`: File identifier returned by Hopen
- `vdata_ref`: Vdata reference number returned by VSattach

### Purpose
Remove a vdata from a file.

### Return value
Returns SUCCEED (or 0) if successful and FAIL (or -1) if not successful.

### Description
VSdelete removes the vdata identified by the parameter `vdata_ref` from the file identified by the parameter `file_id`.

### FORTRAN
```fortran
integer function vsfdlte(file_id, vdata_ref)
  integer file_id, vdata_ref
```

---

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>file_id</code></td>
<td>File identifier returned by Hopen</td>
</tr>
<tr>
<td><code>vdata_ref</code></td>
<td>Vdata reference number returned by VSattach</td>
</tr>
</tbody>
</table>
VSdetach/vsfdtch

int32 VSdetach(int32 vdata_id)

vdata_id IN: Vdata identifier returned by VSattach

Purpose Detaches from the current vdata, terminating further access to that vdata.

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

Description VSdetach detaches from the vdata identified by the parameter vdata_id and updates the vdata information in the file if there are any changes. All memory used for that vdata is freed.

The vdata_id identifier should not be used after that vdata is detached.

FORTRAN integer function vsfdtch(vdata_id)

integer vdata_id
### VSelts/vsfelts

```
int32 VSelts(int32 vdata_id)
```

- **vdata_id**  
  IN: Vdata identifier returned by **VSat**ach

**Purpose**
- Determines the number of records in a vdata.

**Return value**
- Returns the number of records in the vdata if successful and **FAIL** (or -1) otherwise.

**Description**
- **VSelts** returns the number of records in the vdata identified by `vdata_id`.

**FORTRAN**
- ```fortran
  integer function vsfelts(vdata_id)
  integer vdata_id
  ```
VSfdefine/vsffdef

intn VSfdefine(int32 vdata_id, char *fieldname, int32 data_type, int32 order)

vdata_id   IN:  Vdata identifier returned by VSattach
fieldname   IN:  Name of the field to be defined
data_type   IN:  Data type of the field values
order       IN:  Order of the new field

Purpose  Defines a new field for in a vdata.
Return value  Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description  VSfdefine defines a field with the name specified by the parameter fieldname, of the data type specified by the parameter data_type, of the order specified by the parameter order, and within the vdata identified by the parameter vdata_id.

VSfdefine is only used to define fields in a new vdata; it does not set the format of a vdata. Note that defining a field using VSfdefine does not prepare the storage format of the vdata. Once the fields have been defined, the routine VSsetfields must be used to set the format. VSfdefine may only be used with a new empty vdata. Once there is data in a vdata, definitions of vdata fields may not be modified or deleted.

There are certain field names the HDF library recognizes as predefined. A list of these predefined field types can be found in the HDF User’s Guide.

A field is defined by its name (fieldname), its type (data_type) and its order (order). A fieldname is any sequence of characters. By convention, fieldnames are usually a mnemonic, e.g. “PRESSURE”. The type of a field specifies whether a field is float, integer, etc. Thus, data_type may be one of the data types listed in Table 1A in Section I of this manual.

The order of a field is the number of components in that field. A field containing the value of a simple variable, such a time or pressure, would have an order of 1. Compound variables have an order greater than 1. For example, a field containing the values associated with a variable for velocity in three dimensions would have an order of 3.

FORTRAN  integer function vsffdef(vdata_id, fieldname, data_type, order)

integer vdata_id, data_type, order
character*(*) fieldname
VSfexist/vsfex

intn VSfexist(int32 vdata_id, char *field_name_list)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vdata_id</td>
<td>IN: Vdata identifier returned by <strong>VSattach</strong></td>
</tr>
<tr>
<td>field_name_list</td>
<td>IN: List of field names</td>
</tr>
</tbody>
</table>

**Purpose**
Checks to see if certain fields exist in the current vdata.

**Return value**
Returns a value of 1 if all field(s) exist and **FAIL** (or -1) otherwise.

**Description**
**VSfexist** checks if all fields with the names specified in the parameter `field_name_list` exist in the vdata identified by the parameter `vdata_id`.

The parameter `field_name_list` is a string of comma-separated fieldnames (e.g., “PX, PY, PZ” in C and ‘PX, PY, PZ’ in Fortran).

**FORTRAN**

```fortran
integer function vsfex(vdata_id, field_name_list)

integer vdata_id
character*(*) field_name_list
```
VSfind/vsffnd

int32 VSfind(int32 file_id, char *vdata_name)

- **file_id**
  - IN: File identifier returned by **Hopen**
- **vdata_name**
  - IN: Name of the vdata

**Purpose**
Returns the reference number of a vdata, given its name.

**Return value**
Returns the vdata reference number if successful and 0 if the vdata is not found or an error occurs.

**Description**
**VSfind** returns the reference number of the vdata with the name specified by the parameter **vdata_name** in the file specified by the parameter **file_id**. If there is more than one vdata with the same name, **VSfind** will only find the reference number of the first vdata in the file with that name.

**FORTRAN**

```fortran
integer function vsffnd(file_id, vdata_name)

integer file_id
character(*) vdata_name
```
VSfindattr/vsffdat

intn VSfindattr(int32 vdata_id, int32 field_index, char *attr_name)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vdata_id</td>
<td>IN: Vdata identifier returned by VSattach</td>
</tr>
<tr>
<td>field_index</td>
<td>IN: Field index</td>
</tr>
<tr>
<td>attr_name</td>
<td>IN: Attribute name</td>
</tr>
</tbody>
</table>

**Purpose**
Returns the index of an attribute of a vdata or vdata field.

**Return value**
Returns the index of the attribute if successful and FAIL (or -1) otherwise.

**Description**
VSfindattr returns the index of the attribute with the name specified by the parameter attr_name in the vdata identified by the parameter vdata_id.

To return the index of the attribute attached to the vdata, set the value of the parameter field_index to _HDF_VDATA (or -1). To return the index of the attribute of a field of the vdata, set the value of the parameter field_index to the field index. Valid values of field_index range from 0 to the total number of the vdata fields - 1. The number of the vdata fields is returned by VFnfields.

**FORTRAN**
integer function vsffdat(vdata_id, field_index, attr_name)

integer vdata_id, field_index
character*(*) attr_name
VSfindclass/vffcls

int32 VSfindclass(int32 file_id, char *vdata_class)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file_id</td>
<td>IN: File identifier returned by <strong>Hopen</strong></td>
</tr>
<tr>
<td>vdata_class</td>
<td>IN: Class of the vdata</td>
</tr>
</tbody>
</table>

**Purpose**

Returns the reference number of the first vdata with a given vdata class name.

**Return value**

Returns the reference number of the vdata if successful and 0 if the vdata is not found or an error occurs.

**Description**

**VSfindclass** returns the reference number of the vdata with the class name specified by the parameter **vdata_class** in the file identified by the parameter **file_id**.

**FORTRAN**

```fortran
integer function vffcls(vdata_id, vdata_class)

integer vdata_id
character(*) vdata_class
```
VSfindex/vsfidx

intn VSfindex(int32 vdata_id, char *fieldname, int32 *field_index)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vdata_id</td>
<td>Vdata identifier returned by VSattach</td>
</tr>
<tr>
<td>fieldname</td>
<td>Name of the field</td>
</tr>
<tr>
<td>field_index</td>
<td>Index of the field</td>
</tr>
</tbody>
</table>

**Purpose**
Retrieves the index of a field within a vdata.

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

**Description**
VSfindex retrieves the index, field_index, of the field with a name specified by the parameter fieldname, within the vdata identified by the parameter vdata_id.

**FORTRAN**

```fortran
integer function vsfidx(vdata_id, fieldname, field_index)

integer vdata_id, field_index
character*(*) fieldname
```
VSfnattrs/vsffnas

int32 VSfnattrs (int32 \textit{vdata\_id}, int32 \textit{field\_index})

\begin{itemize}
\item \textit{vdata\_id} \textbf{IN:} Vdata identifier returned by \textbf{VSAttach}
\item \textit{field\_index} \textbf{IN:} Index of the field
\end{itemize}

\textbf{Purpose} \quad Returns the number of attributes attached to a vdata or the number of attributes attached to a vdata field.

\textbf{Return value} \quad Returns the number of attributes assigned to this vdata or its fields when successful, and \texttt{FAIL} (or \texttt{-1}) otherwise.

\textbf{Description} \quad \textbf{VSfnattrs} returns the number of attributes attached to a vdata specified by the parameter \textit{vdata\_id}, or the number of attributes attached to a vdata field, specified by the field index, \textit{field\_index}.

To return the number of attributes attached to the vdata, set the value of \textit{field\_index} to \texttt{_HDF\_VDATA} (or \texttt{-1}). To return the number of attributes of a field in the vdata, set the value of \textit{field\_index} to the field index. Field index is a nonnegative integer less than the total number of the vdata fields. The number of vdata fields is returned by \textbf{VFnfields}.

\textbf{VSfnattrs} is different from the \textbf{VSnattrs} routine, which returns the number of attributes of the specified vdata and the fields contained in it.

\textbf{FORTRAN} \quad \texttt{integer function vsffnas(vdata\_id, field\_index)}

\texttt{integer vdata\_id, field\_index}
VSfpack/vsfcpak/vsfnpak

intn VSfpack(int32 vdata_id, intn action, char *fields_in_buf, VOIDP buf, intn buf_size, intn n_records, char *field_name_list, VOIDP bufptrs[])

vdata_id IN: Vdata identifier returned by VSattach
action IN: Action to be performed
fields_in_buf IN: Names of the fields in buf
buf IN/OUT: Buffer containing the values of the packed fields to write to or read from the vdata
buf_size IN: Buffer size in bytes
n_records IN: Number of records to pack or unpack
field_name_list IN: Names of the fields to be packed or unpacked
bufptrs IN/OUT: Array of pointers to the field buffers

Purpose Packs field data into a buffer or unpacks buffered field data into vdata field(s) for fully interlaced fields.

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

Description VSfpack packs or unpacks the field(s) listed in the parameter field_name_list to or from the buffer buf according to the specified action in the parameter action.

Valid values for action are _HDF_VSPACK (or 0) which packs field values from bufptrs (the field buffers) to buf, or _HDF_VSUNPACK (or 1) which unpacks vdata field values from buf into bufptrs.

When VSfpack is called to pack field values into buf, fields_in_buf must list all fields of the vdata. When VSfpack is called to unpack field values, fields_in_buf may be a subset of the vdata fields. To specify all vdata fields in fields_in_buf, NULL can be used in C and a blank character (" ") in Fortran.

The name(s) of the field(s) to be packed or unpacked are specified by the field_name_list. In C, the names in the parameter field_name_list can be a subset of or all field names listed in fields_in_buf. To specify all vdata fields, NULL can be used in C.

The FORTRAN-77 versions of this routine can pack or unpack only one field at a time. Therefore, field_name_list will contain the name of the field that will be packed or unpacked.

The calling program must allocate sufficient space for buf to hold all of the packed fields. The size of the buf buffer should be at least n_records * (the total size of all fields specified in fields_in_buf).
Note that there are two FORTRAN-77 versions of this routine: \texttt{vsfnpak} to pack or unpack a numeric field and \texttt{vsfcpak} to pack or unpack a character field.

Refer to the HDF User's Guide for an example on how to use this routine.

\begin{verbatim}
FORTRAN  
integer function vsfnpak(vdata_id, action, fields_in_buf, buf, 
  buf_size, n_records, field_name_list, bufptrs)
  integer vdata_id, action, buf(*), buf_size, n_records
  character(*) fields_in_buf, field_name_list
  <valid numeric data type> bufptrs(*)

integer function vsfcpak(vdata_id, action, fields_in_buf, buf, 
  buf_size, n_records, field_name_list, bufptrs)
  integer vdata_id, action, buf(*), buf_size, n_records
  character(*) fields_in_buf, field_name_list, bufptrs(*)
\end{verbatim}
VSgetattr/vsfgnat/vsfgcat

intn VSgetattr(int32 vdata_id, intn field_index, int32 attr_index, VOIDP values)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vdata_id</td>
<td>Vdata identifier returned by VSattach</td>
</tr>
<tr>
<td>field_index</td>
<td>Index of the field</td>
</tr>
<tr>
<td>attr_index</td>
<td>Index of the attribute</td>
</tr>
<tr>
<td>values</td>
<td>Buffer for the attribute values</td>
</tr>
</tbody>
</table>

**Purpose**
Retrieves the attribute values of a vdata or vdata field.

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

**Description**
VSgetattr retrieves the attribute values of the vdata identified by the parameter vdata_id or the vdata field specified by the field index, field_index, into the buffer values.

If field_index is set to _HDF_VDATA (or -1), the value of the attribute attached to the vdata is returned. If field_index is set to the field index, attribute attached to a vdata field is returned. Field index is a nonnegative integer less than the total number of the vdata fields. The number of vdata fields is returned by VFnfields.

Attribute to be retrieved is specified by its index, attr_index. Index is a nonnegative integer less than the total number of the vdata or vdata field attributes. Use VSfnattrs to find the number of the vdata or vdata field attributes.

**FORTRAN**

integer function vsfgnat(vdata_id, field_index, attr_index, values)

integer vdata_id, field_index, attr_index
<valid numeric data type> values(*)

integer function vsfgcat(vdata_id, field_index, attr_index, values)

integer vdata_id, field_index, attr_index
character*(*) values
VSgetblockinfo/vsfgetblinfo

intn VSgetblockinfo(int32 vdata_id, int32 *block_size, int32 *num_blocks)

vdata_id IN: Vdata identifier
block_size OUT: Block size in bytes
num_blocks OUT: Number of linked blocks

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

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

Description VSgetblockinfo retrieves the block size and the number of linked blocks for a linked-block Vdata element.

If no response is desired for either returned value, block_size and num_blocks may be set to NULL.

FORTRAN integer function vsfgetblinfo(vdata_id, block_size, num_blocks)

integer vdata_id, num_blocks, block_size
VSgetclass/vsfcls

VSgetclass/vsfcls

int32 VSgetclass(int32 vdata_id, char *vdata_class)

vdata_id  IN:  Vdata identifier returned by VSattach
vdata_class OUT: Vdata class name

Purpose  Retrieves the vdata class name, if any.

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

Description  VSgetclass retrieves the class name of the vdata identified by the parameter vdata_id and places it in the buffer vdata_class.

Space for the buffer vdata_class must be allocated by the calling program before VSgetclass is called. The maximum length of the class name is defined by the macro VSNAMELENMAX (or 64).

FORTRAN

integer function vsfcls(vdata_id, vdata_class)

integer vdata_id

character(*) vdata_class
VSgetfields/vsfgfld

int32 VSgetfields(int32 vdata_id, char *field_name_list)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vdata_id</td>
<td>IN: Vdata identifier returned by VSattach</td>
</tr>
<tr>
<td>field_name_list</td>
<td>OUT: Field name list</td>
</tr>
</tbody>
</table>

Purpose

Retrieves the field names of all of the fields in a Vdata.

Return value

Returns the number of fields in the Vdata if successful and FAIL (or -1) otherwise.

Description

VSgetfields retrieves the names of the fields in the Vdata identified by the parameter vdata_id into the buffer field_name_list.

The parameter field_name_list is a character string containing a comma-separated list of names (e.g., “PX,PY,PZ” in C or ‘PX,PY,PZ’ in Fortran).

The user must allocate the memory space for the buffer field_name_list before calling VSgetfields.

If the Vdata does not have any fields, a null string is returned in the parameter field_name_list.

The maximum length of a Vdata name is defined by VSNAMELENMAX (or 64).

FORTRAN

integer function vsfgfld(vdata_id, field_name_list)

integer vdata_id

character(*) field_name_list
**VSgetid/vsfgid**

```
int32 VSgetid(int32 file_id, int32 vdata_ref)
```

- **file_id**
  - IN: File identifier returned by **Hopen**
- **vdata_ref**
  - IN: Vdata reference number

**Purpose**
Sequentially searches through a file for vdatas.

**Return value**
Returns the reference number for the next vdata if successful and **FAIL** (or -1) otherwise.

**Description**
**VSgetid** sequentially searches through a file identified by the parameter **file_id** and returns the reference number of the next vdata after the vdata that has reference number **vdata_ref**. This routine is generally used to sequentially search the file for vdatas. Searching past the last vdata in a file will result in an error condition.

To initiate a search, this routine must be called with the value of **vdata_ref** equal to -1. Doing so returns the reference number of the first vdata in the file.

**FORTRAN**

```
integer function vsfgid(file_id, vdata_ref)

integer file_id, vdata_ref
```
VSgetinterlace/vsfgint

int32 VSgetinterlace(int32 vdata_id)

vdata_id  IN:  Vdata identifier returned by VSAttach

Purpose  Returns the interlace mode of a vdata.

Return value  Returns FULL_INTERLACE (or 0) or NO_INTERLACE (or 1) if successful and FAIL (or -1) otherwise.

Description  VSgetinterlace returns the interlace mode of the vdata identified by the parameter vdata_id.

FORTRAN  integer function vsfgint(vdata_id)

integer vdata_id
**VSgetname/vsfgnam**

```
int32 VSgetname(int32 vdata_id, char *vdata_name)
```

- **vdata_id**  
  IN: Vdata identifier returned by `VSattach`

- **vdata_name**  
  OUT: Vdata name

**Purpose**  
Retrieves the name of a vdata.

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

**Description**  
`VSgetname` retrieves the name of the vdata identified by the parameter `vdata_id` into the buffer `vdata_name`.

The user must allocate the memory space for the buffer `vdata_name` before calling `VSgetname`. If the vdata does not have a name, a null string is returned in the parameter `vdata_name`. The maximum length of a vdata name is defined by `VSNAMELENMAX` (or 64)

**FORTRAN**

```
integer function vsfgnam(vdata_id, vdata_name)

integer vdata_id
character(*) vdata_name
```
VSgetversion/vsgver

int32 VSgetversion(int32 vdata_id)

vdata_id  IN:  Vdata identifier returned by VSattach

Purpose  Returns the version number of a vdata.

Return value  Returns the version number if successful and FAIL (or -1) otherwise.

Description  VSgetversion returns the version number of the vdata identified by the parameter vdata_id. There are three valid version numbers: VSET_OLD_VERSION (or 2), VSET_VERSION (or 3), and VSET_NEW_VERSION (or 4).

VSET_OLD_VERSION is returned when the vdata is of a version that corresponds to an HDF library version before version 3.2.

VSET_VERSION is returned when the vdata is of a version that corresponds to an HDF library version between versions 3.2 and 4.0 release 2.

VSET_NEW_VERSION is returned when the vdata is of the version that corresponds to an HDF library version of version 4.1 release 1 or higher.

FORTRAN  integer vsgver(vdata_id)

integer vdata_id
VSinquire/VSfinq

intn VSinquire(int32 vdata_id, int32 *n_records, int32 *interlace_mode, char *field_name_list, int32 *vdata_size, char *vdata_name)

vdata_id IN: Vdata identifier returned by VSattach
n_records OUT: Number of records
interlace_mode OUT: Interlace mode of the data
field_name_list OUT: List of field names
vdata_size OUT: Size of a record
vdata_name OUT: Name of the vdata

Purpose Retrieves general information about a vdata.

Return value Returns SUCCEED (or 0) if successful and FAIL (or -1) if it is unable to return any of the requested information.

Description VSinquire retrieves the number of records, the interlace mode of the data, the name of the fields, the size, and the name of the vdata, vdata_id, and stores them in the parameters n_records, interlace_mode, field_name_list, vdata_size, and vdata_name, respectively. In C, if any of the output parameters are NULL, the corresponding information will not be retrieved. Refer to the Reference Manual pages on VSelts, VSgetfields, VSgetinterlace, VSSizeof and VSgetName for other routines that can be used to retrieve specific information.

Possible returned values for interlace_mode are FULL_INTERLACE (or 0) and NO_INTERLACE (or 1). The returned value of vdata_size is the number of bytes in a record and is machine-dependent.

The parameter field_name_list is a character string that contains the names of all the vdata fields, separated by commas. (e.g., “PX,PY,PZ” in C and ‘PX,PY,PZ’ in Fortran).

The user must allocate the memory space for the buffer vdata_name before calling VSinquire. If the vdata does not have a name, a null string is returned in the parameter vdata_name. The maximum length of a vdata name is defined by VSNAMELENMAX (or 64)

FORTRAN  integer function vsfinq(vdata_id, n_records, interlace,
                        field_name_list, vdata_size, vdata_name)

                        integer vdata_id, n_records, interlace, vdata_size
                        character*(*) field_name_list, vdata_name

2-218  December 30, 2003
### VSisattr/vsfisat

```c
intn VSisattr(int32 vdata_id)
```

- **vdata_id**
  - **IN:** Vdata identifier returned by **VSAttach**

### Purpose
Determines whether a vdata is an attribute.

### Return value
Returns `true` (or 1) if the vdata is an attribute, and `false` (or 0) otherwise.

### Description
**VSisattr** determines whether the vdata identified by the parameter `vdata_id` is an attribute.

As attributes are stored by the HDF library as vdatas, a means of testing whether or not a particular vdata is an attribute is needed, and is provided by this routine.

### FORTRAN
```fortran
integer function vsfisat(vdata_id)

integer vdata_id
```
**VSlone/vsflone**

\[ \text{int32 } \text{VSlone}(\text{int32 } \text{file_id}, \text{int32 } \text{ref_array}[], \text{int32 } \text{maxsize}) \]

<table>
<thead>
<tr>
<th><strong>file_id</strong></th>
<th>IN:</th>
<th>File identifier returned by <strong>Hopen</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ref_array</strong></td>
<td>OUT:</td>
<td>Array of reference numbers</td>
</tr>
<tr>
<td><strong>max_refs</strong></td>
<td>IN:</td>
<td>Maximum number of lone vdatas to be retrieved</td>
</tr>
</tbody>
</table>

**Purpose**
Retrieves the reference numbers of all lone vdatas, i.e., vdatas that are not grouped with other objects, in a file.

**Return value**
Returns the total number of lone vdatas if successful and **FAIL** (or -1) otherwise.

**Description**
**VSlone** retrieves the reference numbers of lone vgroups in the file identified by the parameter **file_id**. Although **VSlone** returns the number of lone vdatas in the file, only at most **max_refs** reference numbers are retrieved and stored in the buffer **ref_array**. The array must have at least **max_refs** elements.

An array size of 65,000 integers for **ref_array** is more than adequate if the user chooses to declare the array statically. However, the preferred method is to dynamically allocate memory instead; first call **VSlone** with a value of 0 for **max_refs** to return the total number of lone vdatas, then use the returned value to allocate memory for **ref_array** before calling **VSlone** again.

**FORTRAN**
integer function vsflone(file_id, ref_array, max_refs)

integer file_id, ref_array(*), max_refs
VSnattrs/vsfattrs

```c
int VSnattrs(int32 vdata_id)
```

### Parameters

- **vdata_id**
  - **IN:** Vdata identifier returned by VSattach

### Purpose

Returns the total number of attributes of a vdata and of its fields.

### Return value

Returns the total number of attributes if successful and FAIL (or -1) otherwise.

### Description

**VSnattrs** returns the total number of attributes of the vdata, `vdata_id`, and of its fields.

**VSnattrs** is different from the **Vsfattrs** routine, which returns the number of attributes of a specified vdata or of a field contained in a specified vdata.

### FORTRAN

```fortran
integer function vsfnats(vdata_id)
```

```fortran
integer vdata_id
```
VSread/vsfrd/vsfrdc/vsfread

int32 VSread(int32 vdata_id, uint8 *databuf, int32 n_records, int32 interlace_mode)

- **vdata_id**: IN: Vdata identifier returned by VSattach
- **databuf**: OUT: Buffer to store the retrieved data
- **n_records**: IN: Number of records to be retrieved
- **interlace_mode**: IN: Interlace mode of the data to be stored in the buffer

**Purpose**: Retrieves data from a vdata.

**Return value**: Returns the total number of records read if successful and FAIL (or -1) otherwise.

**Description**

VSread reads \( n_{records} \) records from the vdata identified by the parameter vdata_id and stores the data in the buffer databuf using the interlace mode specified by the parameter interlace_mode.

The user can specify the fields and the order in which they are to be read by calling VSsetfields prior to reading. VSread stores the requested fields in databuf in the specified order.

Valid values for interlace_mode are FULL_INTERLACE (or 1) and NO_INTERLACE (or 0). Selecting FULL_INTERLACE causes databuf to be filled by record and is recommended for speed and efficiency. Specifying NO_INTERLACE causes databuf to be filled by field, i.e., all values of a field in \( n_{records} \) records are filled before moving to the next field. Note that the default interlace mode of the buffer is FULL_INTERLACE.

As the data is stored contiguously in the vdata, VSfpack should be used to unpack the fields after reading. Refer to the discussion of VSfpack in the HDF User’s Guide for more information.

Note that there are three FORTRAN-77 versions of this routine: vsfrd is for buffered numeric data, vsfrdc is for buffered character data and vsfread is for generic packed data.

See the notes regarding the potential performance impact of appendable data sets in the HDF User’s Guide Section 14.4.3, "Unlimited Dimension Data Sets (SDSs and Vdatas) and Performance."

**FORTRAN**

```fortran
integer function vsfrd(vdata_id, databuf, n_records, interlace_mode)

integer vdata_id, n_records, interlace_mode
<valid numeric data type> databuf(*)

integer function vsfrdc(vdata_id, databuf, n_records, interlace_mode)
```
integer vdata_id, n_records, interlace_mode

character*(*) databuf

integer function vsfread(vdata_id, databuf, n_records, interlace_mode)

integer vdata_id, n_records, interlace_mode

integer databuf(*)
### VSseek/vsfseek

#### Purpose
Provides a mechanism for random-access I/O within a vdata.

#### Return value
Returns the record position (zero or a positive integer) if successful and FAIL (or -1) otherwise.

#### Description
**VSseek** moves the access pointer within the vdata identified by the parameter `vdata_id` to the position of the record specified by the parameter `record_pos`. The next call to **VSread** or **VSwrite** will read from or write to the record where the access pointer has been moved to.

The value of `record_pos` is zero-based. For example, to seek to the third record in the vdata, set `record_pos` to 2. The first record position is specified by specifying a `record_pos` value of 0. Each seek is constrained to a record boundary within the vdata.

See the notes regarding the potential performance impact of appendable data sets in the **HDF User’s Guide** Section 14.4.3, "Unlimited Dimension Data Sets (SDSs and Vdatas) and Performance."

#### FORTRAN

```fortran
integer function vsfseek(vdata_id, record_pos)
integer vdata_id, record_pos
```

#### C

```c
int VSseek(int vdata_id, int record_pos)
```
VSsetattr/vsfsnat/vsfscat

intn VSsetattr(int32 vdata_id, int32 field_index, char *attr_name, int32 data_type, int32 count, VOIDP values)

vdata_id IN: Vdata identifier returned by VSattach
field_index IN: Index of the field
attr_name IN: Name of the attribute
data_type IN: Data type of the attribute
count IN: Number of attribute values
values IN: Buffer containing the attribute values

Purpose Sets an attribute of a vdata or a vdata field.

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

Description VSsetattr defines an attribute that has the name specified by the parameter attr_name, the data type specified by the parameter data_type, and the number of values specified by the parameter count, and that contains the values specified in the parameter values. The attribute is set for either the vdata or a vdata field depending on the value of the parameter field_index.

If the field already has an attribute with the same name, the current values will be replaced with the new values if the new data type and order are the same as the current ones. Any changes in the field data type or order will result in a value of FAIL (or -1) to be returned.

If field_index value is set to _HDF_VDATA (or -1), the attribute will be set for the vdata. If field_index is set to the field index, attribute will be set for the vdata field. Field index is a nonnegative integer less than the total number of the vdata fields. The number of vdata fields can be obtained using VFnfields.

The value of the parameter data_type can be any one of the data types listed in Table 1A in Section I of this manual.

FORTRAN integer function vsfsnat(vdata_id, field_index, attr_name, 
data_type, count, values)

integer vdata_id, field_index, data_type, count, values(*)
character*(*) attr_name

integer function vsfscat(vdata_id, field_index, attr_name, 
data_type, count, values)

integer vdata_id, field_index, data_type, count
character*(*) attr_name, values(*)
VSsetblocksize/vsfsetblsz

intn VSsetblocksize(int vdata_id, int block_size)

vdata_id IN: Vdata identifier
block_size IN: Size of each block in bytes

Purpose
Sets linked-block Vdata element block size.

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

Description
VSsetblocksize sets the block size for linked-block elements that will be used to store Vdatas.

The default block size is HDF_APPENDABLE_BLOCK_LEN, which is set to 4096 in the library as it is distributed. VSsetblocksize modifies that default value and must be called before the first write to the Vdata. Once the linked-block element is created, the block size cannot be changed.

The following note may be of interest to users who must pay very close attention to performance issues: VSsetblocksize sets the block size only for blocks following the first block. The first block can be arbitrarily large; the library continues to write to it until it encounters an obstacle, at which point the linked block mechanism is invoked. For example, a Vdata A that is the last item in a file can continue to grow, simply extending the file. If a new Vdata B is then written, that new object is (normally) placed at the end of the file, blocking off extension of the prior Vdata, A. At this point, new writes to A will write data to linked blocks per the block_size and num_blocks settings.

FORTRAN

integer function vsfsetblsz(vdata_id, block_size)

integer vdata_id, block_size
VSsetclass/vsfsc1s

int32 VSsetclass(int32 vdata_id, char *vdata_class)

vdata_id      IN:   Vdata identifier returned by VSattach
vdata_class   IN:   Name of the vdata class

Purpose       Sets the class name of a vdata.

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

Description VSsetclass sets the class name of the vdata identified by the parameter
vdata_id to the value of the parameter vdata_class.

At creation, the class name of a vdata is NULL. The class name may be reset
more than once. Class names, like vdata names, can be any character string.
They exist solely as meaningful labels to user applications and are not used by
the HDF library in any way. Class names will be truncated to VSNAMELENMAX
(or 64) characters.

FORTRAN       integer function vsfsc1s(vdata_id, vdata_class)

integer vdata_id
character*(*) vdata_class
VSsetexternalfile/vsfsextf

VSsetexternalfile/vsfsextf

intn VSsetexternalfile(int32 vdata_id, char *filename, int32 offset)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vdata_id</td>
<td>Vdata identifier returned by VSattach</td>
</tr>
<tr>
<td>filename</td>
<td>Name of the external file</td>
</tr>
<tr>
<td>offset</td>
<td>Offset, in bytes, of the location in the external file the new data is to be written</td>
</tr>
</tbody>
</table>

Purpose
Stores vdata information in an external file.

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

Description
VSsetexternalfile writes data in the vdata identified by the parameter vdata_id in the file named filename, at the byte offset specified by the parameter offset.

Only the data will be stored externally. Attributes and all metadata will remain in the primary HDF file.

IMPORTANT: The user must ensure that the external files are relocated along with the primary file.

Read the Reference Manual page on SDsetexternalfile for more information on using the external file feature.

FORTRAN

integer function vsfsextf(vdata_id, filename, offset)

integer vdata_id, offset
character(*) filename

2-228 December 30, 2003
VSsetfields/vsfsfld

intn VSsetfields(int32 vdata_id, char *field_name_list)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vdata_id</td>
<td>Vdata identifier returned by VSattach</td>
</tr>
<tr>
<td>field_name_list</td>
<td>List of the field names to be accessed</td>
</tr>
</tbody>
</table>

**Purpose**

Specifies the fields to be accessed.

**Return value**

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

**Description**

VSsetfields specifies that the fields, whose names are listed in the parameter field_name_list, of the vdata identified by the parameter vdata_id will be accessed by the next call to VSread or VSwrite. VSsetfields must be called before any call to VSread or VSwrite.

For reading from a vdata, a call to VSsetfields sets up the fields that are to be retrieved from the records in the vdata. If the vdata is empty, VSsetfields will return FAIL (or -1).

For writing to a vdata, VSsetfields can only be called once, to set up the fields in a vdata. Once the vdata fields are set, they may not be changed. Thus, to update some fields of a record after the first write, the user must read all the fields to a buffer, update the buffer, then write the entire record back to the vdata.

The parameter field_name_list is a character string that contains a comma-separated list of fieldnames (i.e., “PX,PY,PZ” in C and 'PX,PY,PZ' in Fortran). The combined width of the fields in a vdata must be less than MAX_FIELD_SIZE (or 65535) bytes. If an attempt to create a larger record is made, VSsetfields will return FAIL (or -1).

If the vdata is attached with an “r” access mode, the parameter field_name_list must contain only the fields that already exist in the vdata. If the vdata is attached with a “w” access mode, field_name_list can contain the names of any fields that have been defined by VSdefine or any predefined fields.

**FORTRAN**

integer function vsfsfld(vdata_id, field_name_list)

integer vdata_id
character(*) field_name_list
VSsetinterlace/vsfsint

intn VSsetinterlace(int32 vdata_id, int32 interlace_mode)

vdata_id  IN:  Vdata identifier returned by VSAttach
interlace_mode  IN:  Interlace mode of the data to be stored in the vdata

Purpose  Sets the interlace mode of a vdata.

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

Description  VSsetinterlace sets the interlace mode of the vdata, vdata_id, to that specified by the parameter interlace_mode. This routine can only be used when creating new vdatas with write access.

The value of interlace_mode may be either FULL_INTERLACE (or 0) or NO_INTERLACE (or 1). If this routine is not called, the default interlace mode of the vdata is FULL_INTERLACE. The FULL_INTERLACE option is more efficient than NO_INTERLACE although both require the same amount of disk space.

Specifying FULL_INTERLACE accesses the vdata by record; in other words, all values of all fields in a record are accessed before moving to the next record. Specifying NO_INTERLACE accesses the vdata by field; in other words, all field values are accessed before moving to the next field. Thus, for writing data, all record data must be available before the write operation is invoked.

Note that the interlace mode of the data to be written is specified by a parameter of the VSWrite routine.

FORTRAN  integer function vsfsint(vdata_id, interlace_mode)

integer vdata_id, interlace_mode
VSsetname/vsfsnam

int32 VSsetname(int32 vdata_id, char *vdata_name)

vdata_id IN: Vdata identifier returned by VSattach
vdata_name IN: Name of the vdata

Purpose Assigns a name to a vdata.
Return value Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description VSsetname sets the name of the vdata identified by the parameter vdata_id to the value of the parameter vdata_name.

At creation, the name of the vdata is NULL. The name may be reset more than once. Vdata names, like class names, can be any character string. They exist solely as a meaningful label for user applications and are not used by the HDF library in any way. Vdata names will be truncated to VSNAMELENMAX (or 64) characters.

FORTRAN integer function vsfsnam(vdata_id, vdata_name)

integer vdata_id
character*(*) vdata_name
VSsetnumblocks/vsfsetnmbl

intn VSsetnumblocks(int32 vdata_id, int32 num_blocks)

vdata_id IN: Vdata identifier
num_blocks IN: Number of blocks to be used for the linked-block element

Purpose
Sets the number of blocks for a linked-block Vdata element.

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

Description
VSsetnumblocks sets the number of blocks in linked-block elements that will be used to store Vdatas.

The default number of blocks is HDF_APPENDABLE_BLOCK_NUM, which is set to 16 in the library as it is distributed. VSsetnumblocks modifies that default value and must be called before the first write to the Vdata. Once the linked-block element is created, the number of blocks cannot be changed.

FORTRAN
integer function vsfsetnmbl(vdata_id, num_blocks)

integer vdata_id, num_blocks
VSsizeof/vsfsiz

int32 VSsizeof(int32 vdata_id, char *field_name_list)

vdata_id  IN: Vdata identifier returned by VSAttach
field_name_list IN: Name(s) of the fields to check

Purpose Computes the size, in bytes, of the given field(s) for the local machine.

Return value Returns the fields size if successful and FAIL (or -1) otherwise.

Description VSsizeof computes the size, in bytes, of the fields specified in the parameter field_name_list in the vdata identified by the parameter vdata_id.

The parameter field_name_list specifies a single field or several comma-separated fields. The field or fields should already exist in the vdata. If more than one field is specified, VSsizeof will return the total sizes of all of the fields.

FORTRAN

integer function vsfsiz(vdata_id, field_name_list)

integer vdata_id
character(*) field_name_list
VWrite/vsfwrt/vsfwrtc/vsfwrit

int32 VWrite(int32 vdata_id, uint8* databuf, int32 n_records, int32 interlace_mode)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vdata_id</td>
<td>IN: Vdata identifier returned by VAttach</td>
</tr>
<tr>
<td>databuf</td>
<td>IN: Buffer of records to be written to the vdata</td>
</tr>
<tr>
<td>n_records</td>
<td>IN: Number of records to be written</td>
</tr>
<tr>
<td>interlace_mode</td>
<td>IN: Interlace mode of the buffer in memory</td>
</tr>
</tbody>
</table>

Purpose

Writes data to a vdata.

Return value

Returns the total number of records written if successful and fails (or -1) otherwise.

Description

VWrite writes the data stored in the buffer databuf into the vdata identified by the parameter vdata_id. The parameter n_records specifies the number of records to be written. The parameter interlace_mode defines the interlace mode of the vdata fields stored in the buffer databuf.

Valid values for interlace_mode are FULL_INTERLACE (or 0) and NO_INTERLACE (or 1). Selecting FULL_INTERLACE fills databuf by record and is recommended for speed and efficiency. Specifying NO_INTERLACE causes databuf to be filled by field, i.e., all values of a field in all records must be written before moving to the next field. Thus, all data must be available before writing. If the data is to be written to the vdata with an interlace mode different from that of the buffer, VSetInterlace must be called prior to VWrite. Note that the default interlace mode of a vdata is FULL_INTERLACE.

It is assumed that the data in databuf is organized as specified by the parameter interlace_mode. The number and order of the fields organized in the buffer must correspond with the number and order of the fields specified in the call to VSetFields, which finalizes the vdata fields definition. Since VWrite writes the data in databuf contiguously to the vdata, VSPack must be used to remove any “padding”, or non-data spaces, used for vdata field alignment. This process is called packing. Refer to the discussion of VSPack in the HDF User’s Guide for more information.

Before writing data to a newly-created vdata, VDefine and VSetFields must be called to define the fields to be written.

Note that there are three FORTRAN-77 versions of this routine: vsfwrt is for buffered numeric data, vsfwrtc is for buffered character data and vsfwrit is for generic packed data.

FORTRAN

integer function vsfwrt(vdata_id, databuf, n_records, interlace_mode)

integer vdata_id, n_records, interlace_mode

<valid numeric data type> databuf(*)
integer function vsfwrtc(vdata_id, databuf, n_records, interlace_mode)

integer vdata_id, n_records, interlace_mode
character(*) databuf

integer function vsfwrit(vdata_id, databuf, n_records, interlace_mode)

integer vdata_id, n_records, interlace_mode
character(*) databuf
DF24addimage/d2aimg

intn DF24addimage(char *filename, VOIDP image, int32 width, int32 height)

filename IN: Name of the file
image IN: Pointer to the image array
width IN: Number of columns in the image
height IN: Number of rows in the image

Purpose
Writes a 24-bit image to the specified file.

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

Description
DF24addimage appends a 24-bit raster image set to the file. Array image is assumed to be width x height x 3 bytes. In FORTRAN-77, the dimensions of the array image must be the same as the dimensions of the image data.

The order in which dimensions are declared is different between C and FORTRAN-77. Ordering varies because FORTRAN-77 arrays are stored in column-major order, while C arrays are stored in row-major order. (Row-major order implies that the last coordinate varies fastest).

When DF24addimage writes an image to a file, it assumes row-major order. The FORTRAN-77 declaration that causes an image to be stored in this way must have the width as its first dimension and the height as its second dimension. In other words, the image must be built “on its side”.

FORTRAN
integer function d2aimg(filename, image, width, height)

character*(*) filename
<valid numeric data type> image
integer width, height
DF24getdims/d2gdims

DF24getdims (char *filename, int32 *width, int32 *height, intn *interlace_mode)

filename IN: Name of the file
width OUT: Width of the image
height OUT: Height of the image
interlace_mode OUT: File interlace mode of the image

Purpose Retrieves dimensions and interlace storage scheme of next image.
Return value Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description DF24getdims retrieves the dimensions and interlace of the image. If the file is being opened for the first time, DF24getdims returns information about the first image in the file. If an image has already been read, DF24getdims finds the next image. In this way, images are read in the same order in which they were written to the file.

If the dimensions and interlace of the image are known beforehand, there is no need to call DF24getdims. Simply allocate arrays with the proper dimensions for the image and invoke DF24getimage to read the images. If, however, you do not know the values of width and height, you must call DF24getdims to get them and then use them to determine the amount of memory to allocate for the image buffer.

Successive calls to DF24getdims and DF24getimage retrieve all of the images in the file in the sequence in which they were written.

The interlace mode codes are: 0 for pixel interlacing, 1 for scan-line interlacing and 2 for scan-plane interlacing.

FORTRAN integer function d2gdims(filename, width, height, interlace_mode)
character*(*) filename
integer width, height, interlace_mode
**DF24getimage/d2gimg**

```c
intn DF24getimage(char *filename, VOIDP image, int32 width, int32 height)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>filename</code></td>
<td>IN: Name of the HDF file</td>
</tr>
<tr>
<td><code>image</code></td>
<td>OUT: Pointer to image buffer</td>
</tr>
<tr>
<td><code>width</code></td>
<td>IN: Number of columns in the image</td>
</tr>
<tr>
<td><code>height</code></td>
<td>IN: Number of rows in the image</td>
</tr>
</tbody>
</table>

**Purpose**
Retrieves an image from the next 24-bit raster image set.

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

**Description**
`DF24getimage` retrieves the image and stores it in an array. If `DF24getdims` has not been called, `DF24getimage` finds the next image in the same way that `DF24getdims` does.

The amount of space allocated for the image should be width x height x 3 bytes.

To specify that the next call to `DF24getimage` should read the raster image using an interlace other than the interlace used to store the image in the file, first call `DF24reqil`.

**FORTRAN**

```fortran
integer function d2gimg(filename, image, width, height)

character(*) filename, image
integer width, height
```
DF24lastref/d2lref

uint16 DF24lastref( )

**Purpose**
Retrieves the last reference number written to or read from a 24-bit raster image set.

**Return value**
Returns the non-zero reference number if successful and FAIL (or -1) otherwise.

**Description**
This routine is primarily used for attaching annotations to 24-bit images and adding 24-bit images to vgroups. DF24lastref returns the reference number of the last 24-bit raster image read or written.

**FORTRAN**
integer function d2lref( )
DF24nimages/d2nimg

intn DF24nimages(char *filename)

filename IN: Name of the file

Purpose Counts the number of 24-bit raster images contained in an HDF file.

Return value Returns the number of 24-bit images in the file if successful and FAIL (or -1) otherwise.

Description DF24nimages counts the number of 24-bit images stored in the file.

FORTRAN integer function d2nimg(filename)

character*(*) filename
DF24putimage/d2pimg

**DF24putimage/d2pimg**

```c
intn DF24putimage(char *filename, VOIDP image, int32 width, int32 height)
```

- `filename` IN: Name of the file
- `image` IN: Pointer to the image array
- `width` IN: Number of columns in the image
- `height` IN: Number of rows in the image

**Purpose**: Writes a 24-bit image as the first image in the file.

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

**Description**: The array image is assumed to be width x height x 3 bytes. **DF24putimage** overwrites any information that exists in the HDF file. To append a new image to a file instead of overwriting an existing file, use **DF24addimage**.

**FORTRAN**

```fortran
integer function d2pimg(filename, image, width, height)

character*(*) filename
<valid numeric data type> image
integer width, height
```
DF24readref/d2rref

intn DF24readref(char *filename, uint16 ref)

filename IN: Name of the file
ref IN: Reference number for the next call to DF24getimage

Purpose Specifies the reference number of the next image to be read when DF24getimage is next called.

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

Description DF24readref is commonly used in conjunction with DFANlablist, which returns a list of labels for a given tag together with their reference numbers. It provides a means of non-sequentially accessing 24-bit raster images in a file.

There is no guarantee that reference numbers appear in sequence in an HDF file. Therefore, it is not safe to assume that a reference number is the index of an image.

FORTRAN integer function d2rref(filename, ref)

character*(*) filename
integer ref
**DF24reqil/d2reqil**

```
intn DF24reqil (intn il)
```

**il**
- **IN**
- Memory interlace of the next image read

**Purpose**
Specifies the interlace mode for the next call to **DF24getimage** will use.

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

**Description**
Regardless of what interlace scheme is used to store the image, **DF24reqil** causes the image to be loaded into memory and be interlaced according to the specification of *il*.

Because a call to **DF24reqil** may require a substantial reordering of the data, slower I/O performance could result than would be achieved if no change in interlace were requested.

The interlace mode codes are: 0 for pixel interlacing, 1 for scan-line interlacing and 2 for scan-plane interlacing.

**FORTRAN**
```
integer function d2reqil(il)

integer il
```
DF24restart/d2first

intn DF24restart( )

**Purpose**
Specifies that the next 24-bit image read from the file will be the first one rather than the 24-bit image following the one most recently read.

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

**FORTRAN**
integer function d2first( )
DF24setcompress/d2scomp

DF24setcompress/d2scomp

intn DF24setcompress(int32 type, comp_info *cinfo)

<table>
<thead>
<tr>
<th>type</th>
<th>IN: Type of compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>cinfo</td>
<td>IN: Pointer to compression information structure</td>
</tr>
</tbody>
</table>

**Purpose**
Set the type of compression to use when writing the next 24-bit raster image.

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

**Description**
This routines provides a method for compressing the next raster image written. The type can be one of the following values: COMP_NONE, COMP_JPEG, COMP_RLE, COMP_IMCOMP. COMP_NONE is the default for storing images if this routine is not called, therefore images are not compressed by default. COMP_JPEG compresses images with a JPEG algorithm, which is a lossy method. COMP_RLE uses lossless run-length encoding to store the image. COMP_IMCOMP uses a lossy compression algorithm called IMCOMP, and is included for backward compatibility only.

The comp_info union contains algorithm-specific information for the library routines that perform the compression and is defined in the hcomp.h header file as follows:

```c
typedef union tag_comp_info
{
    struct
    {
        intn    quality;
        intn    force_baseline;
    } jpeg;

    struct
    {
        int32   nt;
        intn    sign_ext;
        intn    fill_one;
        intn    start_bit;
        intn    bit_len;
    } nbit;

    struct
    {
        intn    skp_size;
    } skphuff;

    struct
    {
        intn    level;
    } deflate;
} comp_info
```
This union is defined to provide future expansion, but is currently only used by the `COMP_JPEG` compression type. A pointer to a valid `comp_info` union is required for all compression types other than `COMP_JPEG`, but the values in the union are not used. The `comp_info` union is declared in the header file `hdf.h` and is shown here for informative purposes only, it should not be re-declared in a user program.

For `COMP_JPEG` compression, the quality member of the jpeg structure must be set to the quality of the stored image. This number can vary from 100, the best quality, to 0, terrible quality. All images stored with `COMP_JPEG` compression are stored in a lossy manner, even images stored with a quality of 100. The ratio of size to perceived image quality varies from image to image, some experimentation may be required to determine an acceptable quality factor for a given application. The `force_baseline` parameter determines whether the quantization tables used during compression are forced to the range 0-255. The `force_baseline` parameter should normally be set to 1 (forcing baseline results), unless special applications require non-baseline images to be used.

If the compression type is JPEG, `d2scomp` defines the default JPEG compression parameters to be used. If these parameters must be changed later, the `d2sjpeg` routine must be used. (See the Reference Manual entry for `d2sjpeg`)

```fortran
integer function d2scomp(type)

integer type
```
### d2scomp

#### integer d2scomp(integer quality, integer baseline)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>quality</td>
<td>IN: JPEG quality specification</td>
</tr>
<tr>
<td>baseline</td>
<td>IN: JPEG baseline specification</td>
</tr>
</tbody>
</table>

**Purpose**: Fortran-specific routine that sets the parameters needed for the JPEG algorithm.

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

**Description**: d8sjpeg changes the JPEG compression parameter settings set in the d8scomp routine.
## d2sjpeg

integer d2sjpeg(integer *quality*, integer *baseline*)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>quality</td>
<td>IN: JPEG quality specification</td>
</tr>
<tr>
<td>baseline</td>
<td>IN: JPEG baseline specification</td>
</tr>
</tbody>
</table>

### Purpose
Fortran-specific routine that sets the parameters needed for the JPEG algorithm.

### Return value
Returns succeed (or 0) if successful and fail (or -1) otherwise.

### Description
*d2sjpeg* changes the JPEG compression parameter settings set in the *d2scomp* routine.
intn DF24setdims(int32 width, int32 height)

**width** IN: Number of columns in the image

**height** IN: Number of rows in the image

**Purpose**
Set the dimensions of the next image to be written to a file.

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

**FORTRAN**

```fortran
integer function d2sdims(width, height)
    integer width, height
```
DF24setil/d2setil

intn DF24setil(intn il)

il
IN: Interlace mode

Purpose
Specifies the interlace mode to be used on subsequent writes.

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

Description
DF24setil sets the interlace mode to be used when writing out the raster image set for a 24-bit image by determining the interlace mode of the image data in memory. If DF24setil is not called, the interlace mode is assumed to be 0.

The interlace mode codes are: 0 for pixel interlacing, 1 for scan-line interlacing and 2 for scan-plane interlacing.

FORTRAN
integer function d2setil(il)

integer il
DFR8addimage/d8aimg

intn DFR8addimage(char *filename, VOIDP image, int32 width, int32 height, uint16 compress)

filename IN: Name of the file
image IN: Array containing the image data
width IN: Number of columns in the image
height IN: Number of rows in the image
compress IN: Type of compression to use, if any

Purpose  Appends the RIS8 for the image to the file.
Return value Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description DFR8addimage is functionally equivalent to DFR8putimage, except that DFR8putimage cannot append image data; it only overwrites.

FORTRAN  integer function d8aimg(filename, image, width, height, compress)

character*(*) filename, image
integer width, height
integer compress
DFR8getdims/d8gdims

intn DFR8getdims(char *filename, int32 *width, int32 *height, intn *ispalette)

filename IN: Name of the HDF file
width OUT: Number of columns in the next image in the file
height OUT: Number of rows in the next image in the file
ispalette OUT: Indicator of the existence of a palette

Purpose
Opens the file, finds the next image, retrieves the dimensions of the image, and determines whether there is a palette associated with the image.

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

Description
DFR8getdims retrieves the dimensions of the image and indicates whether a palette is associated and stored with the image. If the file is being opened for the first time, DFR8getdims returns information about the first image in the file. If an image has already been read, DFR8getdims finds the next image. Thus, images are read in the same order in which they were written to the file.

Normally, DFR8getdims is called before DFR8getimage so that if necessary, space allocations for the image and palette can be checked, and the dimensions can be verified. If this information is already known, DFR8getdims need not be called.

Valid values of ispalette are: 1 if there is a palette, or 0 if not.

FORTRAN
integer function d8gdims(filename, width, height, ispalette)

character(*) filename
integer width, height
integer ispalette
DFR8getimage/d8gimg

intn DFR8getimage(char *filename, uint8 *image, int32 width, int32 height, uint8 *palette)

filename           IN:   Name of the file
image               OUT:  Buffer for the returned image
width               IN:   Width of the image data buffer
height              IN:   Height of the image data buffer
palette             OUT:  Palette data

Purpose               To retrieve the image and its palette, if it is present, and store them in the specified arrays.

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

Description           In C, if palette is NULL, no palette is loaded, even if one is stored with the image. In FORTRAN-77, an array must be allocated to store the palette, even if no palette is expected to be stored. If the image in the file is compressed, DFR8getimage automatically decompresses it. If DFR8getdims has not been called, DFR8getimage finds the next image in the same way that DFR8getdims does.

The width and height parameters specify the number of columns and rows, respectively, in the array which you've allocated in memory to store the image. The image may be smaller than the allocated space.

The order in which you declare dimensions is different between C and FORTRAN-77. Ordering varies because FORTRAN-77 arrays are stored in column-major order, while C arrays are stored in row-major order. (Row-major order implies that the horizontal coordinate varies fastest). When d8gimg reads an image from a file, it assumes row-major order. The FORTRAN-77 declaration that causes an image to be stored in this way must have the width as its first dimension and the height as its second dimension. To take this into account as you read image in your program, the image must be built “on its side”.

FORTRAN

integer function d8gimg(filename, image, width, height, palette)

character*(*) filename, image, palette
integer width, height
DFR8getpalref

intn DFR8getpalref(uint16 *pal_ref)

| pal_ref | OUT: Reference number of the palette |

**Purpose**
Retrieves the reference number of the palette associated with the last image accessed.

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

**Description**
Make certain that **DFR8getdims** is called before **DFR8getpalref**.
DFR8lastref/d8lref

uint16 DFR8lastref( )

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Retrieves the last reference number written to or read from an RIS8.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return value</td>
<td>Returns a non-zero reference number if successful and FAIL (or 0) otherwise.</td>
</tr>
<tr>
<td>Description</td>
<td>This routine is primarily used for attaching annotations to images and adding images to vgroups. <strong>DFR8lastref</strong> returns the reference number of last raster image set read or written.</td>
</tr>
</tbody>
</table>

FORTRAN     
integer function d8lref( )
DFR8nimages/d8nims

intn DFR8nimages(char *filename)

filename IN: Name of the HDF file

Purpose Retrieves the number of 8-bit raster images stored in the specified file.

Return value Returns the number of raster images in the file if successful and FAIL (or -1) otherwise.

FORTRAN integer function d8nims(filename)
character*(*) filename
DFR8putimage/d8pimg

intn DFR8putimage(char *filename, VOIDP image, int32 width, int32 height, uint16 compress)

filename IN: Name of the file to store the raster image in
image IN: Array with image to put in file
width IN: Number of columns in the image
height IN: Number of rows in the image
compress IN: Type of compression used, if any

Purpose
 Writes the RIS8 for the image as the first image in the file, overwriting any information previously in the file.

Return value
 Returns succeed (or 0) if successful and fail (or -1) otherwise.

Description
 The compress parameter identifies the method to be used for compressing the data, if any. If IMCOMP compression is used, the image must include a palette.

DFR8putimage overwrites any information that exists in the HDF file. To write an image to a file by appending it, rather than overwriting it, use DFR8addimage.

In FORTRAN-77, the dimensions of the image array must be the same as the dimensions of the image itself.

The order in which dimensions are declared is different between C and FORTRAN-77. Ordering varies because FORTRAN-77 arrays are stored in column-major order, while C arrays are stored in row-major order. (Row-major order implies that the horizontal coordinate varies fastest). When DFR8putimage writes an image to a file, it assumes row-major order. The FORTRAN-77 declaration that causes an image to be stored in this way must have the width as its first dimension and the height as its second dimension, the reverse of the way it is done in C. To take this into account as you build your image in your FORTRAN-77 program, the image must be built “on its side”.

FORTRAN

integer function d8pimg(filename, image, width, height, compress)

character(*) filename, image

integer width, height, compress
DFR8readref/d8rref

DFR8readref/d8rref

intn DFR8readref(char *filename, uint16 ref)

filename IN: Name of the file

ref IN: Reference number for next DFR8getimage

Purpose Specifies the reference number of the image to be read when DFR8getimage is next called.

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

Description DFR8readref is usually used in conjunction with DFANlablist, which returns a list of labels for a given tag together with their reference numbers. It provides, in a sense, a random access to images. There is no guarantee that reference numbers appear in sequence in an HDF file; therefore, it is not safe to assume that a reference number is the index of an image.

FORTRAN integer function d8rref(filename, ref)

character*(*) filename

integer ref
DFR8restart/d8first

intn DFR8restart( )

**Purpose**
Causes the next get command to read from the first raster image set in the file.

**Return value**
Returns **succeed** (or 0) if successful and **fail** (or -1) otherwise.

**FORTRAN**
integer function d8first( )
```c
intn DFR8setcompress(int32 type, comp_info *cinfo)
```

**type**

IN: Type of compression

**cinfo**

IN: Pointer to compression information structure

**Purpose**

Sets the compression type to be used when writing the next 8-bit raster image.

**Return value**

Returns **SUCCEED** (or 0) if successful and **FAIL** (or -1) otherwise.

**Description**

This routine provides a method for compressing the next raster image written. The type can be one of the following values: **COMP_NONE**, **COMP_JPEG**, **COMP_RLE**, **COMP_IMCOMP**. **COMP_NONE** is the default for storing images if this routine is not called, therefore images are not compressed by default. **COMP_JPEG** compresses images with a JPEG algorithm, which is a lossy method. **COMP_RLE** uses lossless run-length encoding to store the image. **COMP_IMCOMP** uses a lossy compression algorithm called IMCOMP, and is included for backward compatibility only.

The `comp_info` union contains algorithm-specific information for the library routines that perform the compression and is defined in the `hcomp.h` header file as follows (refer to the header file for inline documentation):

```c
typedef union tag_comp_info
{
    struct
    {
        intn    quality;
        intn    force_baseline;
    } jpeg;

    struct
    {
        int32   nt;
        intn    sign_ext;
        intn    fill_one;
        intn    start_bit;
        intn    bit_len;
    } nbit;

    struct
    {
        intn    skp_size;
    } skphuff;

    struct
    {
        intn    level;
    } deflate;
} comp_info;
```
This union is defined to provide future expansion, but is currently only used by the \texttt{COMP\_JPEG} compression type. A pointer to a valid \texttt{comp\_info} union is required for all compression types other than \texttt{COMP\_JPEG}, but the values in the union are not used. The \texttt{comp\_info} union is declared in the header file \texttt{hdf.h} and is shown here for informative purposes only, it should not be re-declared in a user program.

For \texttt{COMP\_JPEG} compression, the quality member of the jpeg structure must be set to the quality of the stored image. This number can vary from 100, the best quality, to 0, terrible quality. All images stored with \texttt{COMP\_JPEG} compression are stored in a lossy manner, even images stored with a quality of 100. The ratio of size to perceived image quality varies from image to image, some experimentation may be required to determine an acceptable quality factor for a given application. The \texttt{force\_baseline} parameter determines whether the quantization tables used during compression are forced to the range 0–255. It should normally be set to 1 (forcing baseline results), unless special applications require non-baseline images to be used.

If the compression type is JPEG, \texttt{d8scomp} defines the default JPEG compression parameters to be used. If these parameters must be changed later, the \texttt{d8sjpeg} routine must be used. (Refer to the Reference Manual page on \texttt{d8sjpeg}).

\begin{verbatim}
FORTRAN

integer function d8scomp(type)

integer type

\end{verbatim}
integer d8scomp(integer quality, integer baseline)

**Purpose**
Fortran-specific routine that sets the parameters needed for the JPEG algorithm.

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

**Description**
d8sjpeg changes the JPEG compression parameter settings set in the d8scomp routine.
**d8sjpeg**

`integer d8sjpeg(integer quality, integer baseline)`

- **quality** IN: JPEG quality specification
- **baseline** IN: JPEG baseline specification

**Purpose**
Fortran-specific routine that sets the parameters needed for the JPEG algorithm.

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

**Description**
`d8sjpeg` changes the JPEG compression parameter settings set in the `d8scomp` routine.
DFR8setpalette/d8spal

intn DFR8setpalette(uint8 *palette)

palette IN: Palette data

Purpose Indicate which palette, if any, is to be used for subsequent image sets.

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

Description The specified palette remains the default palette until changed by a subsequent call to DFR8setpalette.

FORTRAN integer function d8spal(palette)

character(*) palette
DFR8writeref/d8wref

intn DFR8writeref(char *filename, uint16 ref)

filename  IN: Name of the HDF file
ref  IN: Reference number for next call to DFR8putimage or DFR8addimage

Purpose  Specifies the reference number of the image to be written when DFR8addimage or DFR8putimage is next called.

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

Description  It is unlikely that you will need this routine, but if you do, use it with caution. There is no guarantee that reference numbers appear in sequence in an HDF file; therefore, it is not safe to assume that a reference number is the index of an image. In addition, using an existing reference number will overwrite the existing 8-bit raster image data.

FORTRAN  integer function d8wref(filename, ref)

character*(*) filename
integer ref
DFPaddpal/dpapel

intn DFPaddpal(char *filename, VOIDP palette)

filename IN: Name of the HDF file
palette IN: Buffer containing the palette to be written

Purpose Appends a palette to a file.

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

Description If the named file does not exist, it is created and the palette written to it. The palette buffer should be at least 768 bytes in length.

FORTRAN integer function dpapel(filename, palette)

character*(*) filename, palette
DFPgetpal/dpgpal

intn DFPgetpal(char *filename, VOIDP palette)

\textit{filename} \hspace{1cm} \textbf{IN:} \hspace{1cm} Name of the HDF file

\textit{palette} \hspace{1cm} \textbf{OUT:} \hspace{1cm} Buffer for the returned palette

\textbf{Purpose} \hspace{3.5cm} Retrieves the next palette from file and stores it in the buffer \textit{palette}.

\textbf{Return value} \hspace{3.5cm} Returns \texttt{SUCCEED} (or 0) if successful and \texttt{FAIL} (or -1) otherwise.

\textbf{Description} \hspace{3.5cm} The \textit{palette} buffer is assumed to be at least 768 bytes long. Successive calls to \texttt{DFPgetpal} retrieve the palettes in the sequence they are stored in the file.

\texttt{FORTRAN} \hspace{3.5cm} integer function dpgpal(filename, palette)

\hspace{3cm} character*(*) filename, palette
DFPlastref/dplref

uint16 DFPlastref(void)

Purpose
Returns the value of the reference number most recently read or written by a palette function call.

Return value
Returns the reference number if successful and FAIL (or -1) otherwise.

FORTRAN
integer function dplref( )
DFPnpals/dpnlpals

DFPnpals/dpnlpals

intn DFPnpals(char *filename)

filename IN: Name of the file

Purpose Indicates the number of palettes in the specified file.

Return value Returns the number of palettes if successful and FAIL (or -1) otherwise.

FORTRAN integer function dpnlpals(filename)

character*(*) filename
DFPputpal/dpppal

intn DFPputpal (char *filename, VOIDP palette, intn overwrite, char *filemode)

filename IN: Name of the file
palette IN: Buffer containing the palette to be written
overwrite IN: Flag identifying the palette to be written
filemode IN: File access mode

Purpose: Writes a palette to the file.
Return value: Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description: This routine provides more control of palette write operations than DFPaddpal. Note that the combination filemode="w" and overwrite=1 has no meaning and will result in an error condition. To overwrite a palette, filename must be the same filename as the last file accessed through the DFP interface.

Valid values for overwrite are: 1 to overwrite last palette; 0 to write a new palette.

Valid values for filemode are: “a” to append the palette to the file and “w” to create a new file.

The palette buffer must be at least 768 bytes in length.

FORTRAN

integer function dpppal(filename, palette, overwrite, filemode)

ccharacter(*) filename, palette, filemode
integer overwrite
DFPreadref/dprref

DFPreadref/dprref

intn DFPreadref(char *filename, uint16 ref)

filename  IN:  Name of the file
ref        IN:  Reference number to be used in next DFPgetpal call

Purpose  Retrieves the reference number of the palette to be retrieved next by DFPgetpal.

Return value  Returns SUCCEED (or 0) if the palette with the specified reference number exists and FAIL (or -1) otherwise.

Description  Used to set the reference number of the next palette to be retrieved.

FORTRAN  

integer function dprref(filename, ref)

character(*) filename
integer ref
DFPrestart/dprest

intn DFPrestart( )

**Purpose**
Specifies that **DFPgetpal** will read the first palette in the file, rather than the next unread palette.

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

**FORTRAN**
integer function dprest( )
DFPwriteref/dpwref

DFPwriteref/dpwref

intn DFPwriteref(char *filename, uint16 ref)

    filename IN:   Name of the file
    ref    IN:   Reference number to be assigned to the next palette written to a file

Purpose     Determines the reference number of the next palette to be written.
Return value Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description The file name is ignored. The next palette written, regardless of the filename, is assigned the reference number ref.

FORTRAN    integer function dpwref(filename, ref)

            character*(*) filename
            integer ref
DFKNTsize

int DFKNTsize(int32 data_type)

\[data\_type\]  \hspace{1em} \text{IN: Data type}

**Purpose**  
Determines the size of the specified data type.

**Return value**  
Returns the size, in bytes, of the specified data type if successful and \text{FAIL} (or \text{-1}) otherwise.
**DFUfptoiimage/duf2im**

```fortran
int DFUfptoiimage(int32 hdim, int32 vdim, float32 max, float32 min, float32 *hscale, float32 *vscale, float32 *data, uint8 *palette, char *outfile, int ct_method, int32 hres, int32 vres, int compress)
```

- **hdim** IN: Horizontal dimension of the input data
- **vdim** IN: Vertical dimension of the input data
- **max** IN: Maximum value of the input data
- **min** IN: Minimum value of the input data
- **hscale** IN: Horizontal scale of the input data (optional)
- **vscale** IN: Vertical scale of the input data (optional)
- **data** IN: Buffer containing the input data
- **palette** IN: Pointer to the palette data
- **outfile** IN: Name of the file the image data will be stored in
- **ct_method** IN: Color transformation method
- **hres** IN: Horizontal resolution to be applied to the output image
- **vres** IN: Vertical resolution to be applied to the output image
- **compress** IN: Compression flag

**Purpose**
Converts floating point data to 8-bit raster image format and stores the converted image data in the specified file.

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

**Description**
This routine is very similar to the utility `fptohdf`, which takes its input from one or more files, rather than from internal memory. Another difference is that this routine allows compression (run-length encoding), whereas `fptohdf` does not at present.

As this routine is meant to mimic many of the features of NCSA DataScope, much of the code has been taken directly from the DataScope source.

Valid values for **ct_method** are: 1 (or **EXPAND**) for expansion and 2 (or **INTERP**) for interpolation.

Valid values for **compress** are: 0 for no compression and 1 for compression enabled.

**FORTRAN**
```fortran
integer function duf2im(hdim, vdim, max, min, hscale, vscale, data, palette, outfile, ct_method, hres, vres, compress)
integer hdim, vdim
```
real max, min, hscale, vscale, data
character*(*) palette, outfile
integer ctmethod, hres, vres, compress
DFANaddfds/daafds

intn DFANaddfds(int32 file_id, char *description, int32 desc_len)

    file_id  IN:  File identifier returned by Hopen
    description  IN:  Sequence of ASCII characters (may include NULL or '\0')
    desc_len  IN:  Length of the description

Purpose  Adds a file description to a file.
Return value  Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description  These annotations are associated with the file, not with any particular object within the file. The parameter description can contain any sequence of ASCII characters. It does not have to be a string. Use the general purpose routines Hopen and Hclose to manage file access as the file annotation routines will not open and close HDF files.

FORTRAN  integer function daafds(file_id, description, desc_len)

        integer file_id, desc_len
        character(*) description
DFANaddfid/daafid

DFANaddfid/daafid

intn DFANaddfid(int32 file_id, char *label)

file_id IN: The file identifier returned by Hopen.
label IN: A null-terminated string.

Purpose Writes a file label to a file.

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

Description These annotations are associated with the file, not with any particular object within the file. The label must be a single string. Use the general purpose routines Hopen and Hclose to manage file access because the file annotation routines will not open and close HDF files for you.

In the FORTRAN-77 version, the string length for the label should be close to the actual expected string length, because in FORTRAN-77 string lengths generally are assumed to be the declared length of the array that holds the string.

FORTRAN

integer function daafid(file_id, label)

integer file_id
character*(*) label
DFANclear/daclear

intn DFANclear( )

**Purpose**
Resets all internal library structures and parameters of the DFAN annotation interface.

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

**Description**
When a file is regenerated in a single run by a library routine of another interface (such as DFSDputdata), DFANclear should be called to reset the interface.

**FORTRAN**
integer function daclear( )
DFANgetdesc/dagdesc

DFANgetdesc/dagdesc

intn DFANgetdesc(char *filename, uint16 tag, uint16 ref, char *desc_buf, int32 buf_len)

**filename** IN: Name of the file
**tag** IN: Tag of the data object assigned the description
**ref** IN: Reference number of the data object assigned the description
**desc_buf** OUT: Buffer allocated to hold the description
**buf_len** IN: Size of the buffer allocated to hold the description

**Purpose**
Reads the description assigned to the data object with the given tag and reference number.

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

**Description**
The parameter **buf_len** specifies the storage space available for the description. The length of **buf_len** must account for the null termination character appended to the description.

**FORTRAN**

integer function dagdesc(filename, tag, ref, desc_buf, buf_len)
character*(*) filename, desc_buf
integer tag, ref
integer buf_len
DFANgetdesclen/dagdlen

int32 DFANgetdesclen(char *filename, uint16 tag, uint16 ref)

filename IN: Name of the file
tag IN: Tag of the data object assigned the description
ref IN: Reference number of the data object assigned the description

Purpose
Retrieves the length of a description of the data object with the given tag and reference number.

Return value
Returns the length of a description if successful and FAIL (or -1) otherwise.

Description
This routine should be used to insure that there is enough space allocated for a description before actually reading it.

FORTRAN
integer function dagdlen(filename, tag, ref)

character*(*) filename
integer tag, ref
DFANgetfds/dagfds

int32 DFANgetfds(int32 file_id, char *desc_buf, int32 buf_len, intn isfirst)

file_id IN: File identifier returned by Hopen
desc_buf OUT: The buffer allocated to hold the description
buf_len IN: Size of the buffer allocated to hold the description
isfirst IN: Determines the description to be retrieved

Purpose
Reads the next file description.

Return value
Returns the length of the file description if successful and FAIL (or -1) otherwise.

Description
If isfirst is 0, DFANgetfds gets the next file description from an HDF file. For example, if there are three file descriptions in a file, three successive calls to DFANgetfds will get all three descriptions. If isfirst is 1, DFANgetfds gets the first file description.

Valid values for isfirst are: 1 to read the first description and 0 to read the next description.

FORTRAN
integer function dagfds(file_id, desc_buf, buf_len, isfirst)

integer file_id, buf_len, isfirst
classerAwesome{(*) desc_buf
DFANgetfdslen/dagfdsl

```
int32 DFANgetfdslen(int32 file_id, intn isfirst)
```

- **file_id**
  - IN: File identifier returned by `Hopen`
- **isfirst**
  - IN: Determines the description the retrieved length information applies to

**Purpose**: Returns the length of a file description.

**Return value**: Returns the length of the file description if successful and `FAIL` (or `-1`) otherwise.

**Description**: When `DFANgetfdslen` is first called for a given file, it returns the length of the first file description. In order to get the lengths of successive file descriptions, you must call `DFANgetfds` between calls to `DFANgetfdslen`. Successive calls to `DFANgetfdslen` without calling `DFANgetfds` between them will return the length of the same file description.

Valid values for `isfirst` are: 1 to read the length of the first description and 0 to read the length of the next description.

**FORTRAN**
```
integer function dagfdsl(file_id, isfirst)
```

```
integer file_id, isfirst
```
DFANgetfid/dagfid

DFANgetfid/dagfid

int32 DFANgetfid(int32 file_id, char *desc_buf, int32 buf_len, intn isfirst)

file_id IN: File identifier returned by Hopen
label_buf OUT: The buffer allocated to hold the label
buf_len IN: Size of the buffer allocated to hold the label
isfirst IN: Determines the file label to be retrieved

Purpose Reads a file label from a file.
Return value Returns the length of the file description if successful and FAIL (or -1) otherwise.
Description If isfirst is 0, DFANgetfid gets the next file label from the file. If isfirst is 1, DFANgetfid gets the first file label in the file. If buf_len is not large enough, the label is truncated to buf_len-1 characters in the buffer label_buf.
Valid values of isfirst are: 1 to read the first label, 0 to read the next label

FORTRAN integer function dagfid(file_id, label_buf, buf_len, isfirst)

integer file_id, buf_len, isfirst
character*(*) label_buf
DFANgetfidlen/dagfidl

int32 DFANgetfidlen(int32 file_id, intn isfirst)

<table>
<thead>
<tr>
<th>file_id</th>
<th>IN:</th>
<th>File identifier returned by Hopen</th>
</tr>
</thead>
<tbody>
<tr>
<td>isfirst</td>
<td>IN:</td>
<td>Determines the file label the retrieved length information applies to</td>
</tr>
</tbody>
</table>

**Purpose**

Returns the length of a file label.

**Return value**

Returns the length of the file label if successful and FAIL (or -1) otherwise.

**Description**

When DFANgetfidlen is first called for a given file, it returns the length of the first file label. In order to retrieve the lengths of successive file labels, DFANgetfid must be called between calls to DFANgetfidlen. Otherwise, successive calls to DFANgetfidlen will return the length of the same file label.

Valid values of isfirst are: 1 to read the first label, and 0 to read the next label.

**FORTRAN**

integer function dagfidl(file_id, isfirst)

integer file_id, isfirst
DFANgetlabel/daglab

DFANgetlabel(daglab

intn DFANgetlabel(char *filename, uint16 tag, uint16 ref, char *label_buf, int32 buf_len)

filename IN: Name of the HDF file
tag IN: Tag of the data object assigned the label
ref IN: Reference number of the data object assigned the label
label_buf OUT: Buffer for the label
buf_len IN: Size of the buffer allocated for the label

Purpose
Reads the label assigned to the data object identified by the given tag and reference number.

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

Description
The parameter buf_len specifies the storage space available for the label. The length of buf_len must account for the null termination character appended to the annotation.

FORTRAN

integer function daglab(filename, tag, ref, label_buf, buf_len)

character*(*) filename, label_buf
integer tag, ref, buf_len
DFANgetlablen/dagllen

int32 DFANgetlablen(char *filename, uint16 tag, uint16 ref)

filename  IN: Name of the file

tag        IN: Tag of the data object assigned the label

ref        IN: Reference number the data object assigned the label

Purpose    Returns the length of a label assigned to the object with a given tag and reference number.

Return value Returns the length of the label if successful and FAIL (or -1) otherwise.

Description This routine should be used to insure that there is enough space allocated for a label before actually reading it.

FORTRAN    integer function dagllen(filename, tag, ref)

character*(*) filename

integer tag, ref
DFANlablist/dallist

DFANlablist/dallist

int DFANlablist(char *filename, uint16 tag, unit16 ref_list[], char *label_list, int list_len, intn label_len, intn start_pos)

filename  IN: Name of the file
tag  IN: Tag to be queried
ref_list  OUT: Buffer for the returned reference numbers
label_list  OUT: Buffer for the returned labels
list_len  IN: Size of the reference number list and the label list
label_len  IN: Maximum length allowed for a label
start_pos  IN: Starting position of the search

Purpose
Returns a list of all reference numbers and labels (if labels exist) for a given tag.

Return value
Returns the number of reference numbers found if successful and FAIL (or -1) otherwise.

Description
Entries are returned from the start_pos entry up to the list_len entry.

The list_len determines the number of available entries in the reference number and label lists, label_len is the maximum length allowed for a label, and start_pos tells which label to start reading for the given tag. (If start_pos is 1, for instance, all labels will be read; if start_pos is 4, all but the first 3 labels will be read.) The ref_list contains a list of reference numbers for all objects with a given tag. The label_list contains a corresponding list of labels, if any. If there is no label stored for a given object, the corresponding entry in label_list is an empty string.

Taken together, the ref_list and label_list constitute a directory of all objects and their labels (where they exist) for a given tag. The label_list parameter can display all of the labels for a given tag. Or it can be searched to find the reference number of a data object with a certain label. Once the reference number for a given label is found, the corresponding data object can be accessed by invoking other HDF routines. Therefore, this routine provides a mechanism for the direct access to data objects in HDF files.

FORTRAN

integer function dallist(filename, tag, ref_list, label_list, list_len, label_len, start_pos)

character(*) filename, label_list
integer ref_list(*)
integer list_len, label_len, start_pos
DFANlastref/dalref

uint16 DFANlastref(  )

**Purpose**
Returns the reference number of the annotation last written or read.

**Return value**
Returns the reference number if successful and FAIL (or -1) otherwise.

FORTRAN
integer function dalref(  )
DFANputdesc/dapdesc

int DFANputdesc(char *filename, uint16 tag, uint16 ref, char *description, int32 desc_len)

filename IN: Name of the file
tag IN: Tag of the data object to be assigned the description
ref IN: Reference number the data object to be assigned the description
description IN: Sequence of ASCII characters (may include NULL or \0)
desc_len IN: Length of the description

Purpose Writes a description for the data object with the given tag and reference number.

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

Description The parameter description can contain any sequence of ASCII characters; it does not have to be a string. If DFANputdesc is called more than once for the same tag/reference number pair, only the last description is stored in the file.

FORTRAN integer function dapdesc(filename, tag, ref, description, desc_len)

character*(*) filename, description
integer tag, ref, desc_len
DFANputlabel/daplab

intn DFANputlabel(char *filename, uint16 tag, uint16 ref, char *label)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>Name of the file</td>
</tr>
<tr>
<td>tag</td>
<td>Tag of the data object to be assigned the label</td>
</tr>
<tr>
<td>ref</td>
<td>Reference number the data object to be assigned the label</td>
</tr>
<tr>
<td>label</td>
<td>Null-terminated label string</td>
</tr>
</tbody>
</table>

**Purpose**
Assigns a label to the data object with the given tag/reference number pair.

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

**FORTRAN**
integer function daplab(filename, tag, ref, label)

character*(*) filename, label
integer tag, ref
DFSDadddata/dsadata

intn DFSDadddata(char *filename, intn rank, int32 dimsizes[], VOIDP data)

filename IN: Name of the HDF file
rank IN: Number of dimensions in the data array to be written
dimsizes IN: Array containing the size of each dimension
data IN: Array containing the data to be stored

Purpose
Appends a scientific dataset in its entirety to an existing HDF file if the file
exists. If not, a new file is created.

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

Description
In addition to appending a multidimensional array of data to an HDF file,
DFSDadddata automatically stores any information pertinent to the dataset. It
will not overwrite existing data in the file. The array data can be of any valid
type. However, if no data type has been set by DFSDsetNT, it is assumed that
the data is of type float32.

Calling DFSDadddata will write the scientific dataset and all associated
information. That is, when DFSDadddata is called, any information set by a
DFSDset* call is written to the file, along with the data array itself.

FORTRAN
integer function dsadata(filename, rank, dimsizes, data)

character*(*) filename
integer rank
integer dimsizes(*), data(*)
### DFSDclear/dsclear

intn DFSDclear( )

<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Clears all values set by DFSDset* routines.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Return value</strong></td>
<td>Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>After a call to DFSDclear, values set by any DFSDset* call will not be written unless they have been set again.</td>
</tr>
</tbody>
</table>

FORTRAN integer function dsclear( )
**DFSDendslab/dseslab**

```fortran
integer function dseslab( )
```

**Purpose**
Terminates a sequence of slab calls started by **DFSDstartslab** by closing the file.

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

---

**FORTRAN**
integer function dseslab( )
DFSDendslice/dseslc

intn DFSDendslice( )

**Purpose**
Terminates the write operation after storing a slice of data in a scientific dataset.

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

**Description**
DFSDendslice must be called after all the slices are written. It checks to ensure that the entire dataset has been written, and if it has not, returns an error code. DFSDendslice is obsolete in favor of DFSDendslab. DFSDendslab is the recommended function call to use when terminating hyperslab (previously known as data slices) operations. HDF will continue to support DFSDendslice only to maintain backward compatibility with earlier versions of the library.

**FORTRAN**
integer function dseslc( )
DFSDgetcal/dsgcal

int32 DFSDgetcal(float64 *cal, float64 *cal_err, float64 *offset, float64 *offset_err, int32 *data_type)

cal OUT: Calibration factor

offset OUT: Uncalibrated offset

offset_err OUT: Uncalibrated offset error

data_type OUT: Data type of uncalibrated data

Purpose
Retrieves the calibration record, if there is one, attached to a scientific dataset.

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

Description
A calibration record contains four 64-bit floating point values followed by a 32-bit integer.

The relationship between a value iy stored in a dataset and the actual value y is defined as:

\[ y = \text{cal} \times (iy - \text{offset}) \]

The variable offset_err contains a potential error of offset, and cal_err contains a potential error of cal.
Currently the calibration record is provided for information only. The SD interface performs no operations on the data based on the calibration tag.

As an example, suppose the values in the calibrated dataset iy[] are the following integers:

\[ iy[6] = \{2, 4, 5, 11, 26, 81\} \]

By defining cal = 0.50 and offset = -200.0 and applying the calibration formula, the calibrated dataset iy[] returns to its original form as a floating point array:

\[ y[6] = \{1001.0, 1002.0, 1002.5, 1005.5, 1013.0, 1040.5\} \]

FORTRAN

integer function dsgcal(cal, cal_err, offset, offset_err, data_type)

real cal, cal_err, offset, offset_err

integer data_type
**DFSDgetdata/dsgdata**

```c
intn DFSDgetdata(char *filename, intn rank, int32 dimsizes[], VOIDP data)
```

- **filename** IN: Name of the file
- **rank** IN: Number of dimensions
- **dimsizes** IN: Dimensions of the data buffer
- **data** OUT: Buffer for the data

**Purpose**: Reads the next dataset in the file.

**Return value**: Returns `SUCCEED` (or 0) if successful and `FAIL` (or -1) otherwise.

**Description**: If the values of `rank` or `dimsizes` aren’t known, `DFSDgetdims` must be called to retrieve them and then use them to determine the buffer space needed for the array data. If the data type of the data in a scientific dataset isn’t know, `DFSDgetNT` must be called to retrieve it. Subsequent calls to `DFSDgetdata` (or to `DFSDgetdims` and `DFSDgetdata`) will sequentially read scientific datasets from the file. For example, if `DFSDgetdata` is called three times in succession, the third call reads data from the third scientific dataset in the file.

If `DFSDgetdims` or `DFSDgetdata` is called and there are no more scientific datasets left in the file, an error code is returned and nothing is read. `DFSDrestart` can be used to override this convention.

**FORTRAN**

```fortran
integer function dsgdata(filename, rank, dimsizes, data)

character(*) filename
integer rank
integer dimsizes(*), data(*)
```
DFSDgetdatalen/dsgdaln

intn DFSDgetdatalen(intn *label_len, intn *unit_len, intn *format_len, intn *coords_len)

<table>
<thead>
<tr>
<th>label_len</th>
<th>OUT:</th>
<th>Maximum length of the label string</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit_len</td>
<td>OUT:</td>
<td>Maximum length of the unit string</td>
</tr>
<tr>
<td>format_len</td>
<td>OUT:</td>
<td>Maximum length of the format string</td>
</tr>
<tr>
<td>coords_len</td>
<td>OUT:</td>
<td>Maximum length of the coordinate system string</td>
</tr>
</tbody>
</table>

Purpose

Retrieves the lengths of the label, unit, format, and coordinate system strings.

Return value

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

Description

The space allocated for the label, unit, format, and coordinate system strings must be at least one byte larger than the actual length of the string to account for the null termination.

FORTRAN

integer function dsgdaln(label_len, unit_len, format_len, coords_len)

integer label_len, unit_len, format_len, coords_len
DFSDgetdatastrs/dsgdast

Purpose
Retrieves information about the label, unit, and format attribute strings associated with the data.

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

Description
The parameter coordsys gives the coordinate system that is to be used for interpreting the dimension information.

FORTRAN
integer function dsgdast(label, unit, format, coordsys)
character(*) label, unit, format, coordsys

DFSDgetdatastrs/dsgdast

intn DFSDgetdatastrs(char *label, char *unit, char *format, char *coordsys)

label OUT: Label describing the data
unit OUT: Unit to be used with the data
format OUT: Format to be used in displaying data
coordsys OUT: Coordinate system
DFSDgetdimlen/dsgdiln

intn DFSDgetdimlen (intn *dim, intn *label_len, intn *unit_len, intn *format_len)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dim</td>
<td>Dimension the label, unit, and format refer to</td>
</tr>
<tr>
<td>label_len</td>
<td>OUT: Length of the label</td>
</tr>
<tr>
<td>unit_len</td>
<td>OUT: Length of the unit</td>
</tr>
<tr>
<td>format_len</td>
<td>OUT: Length of the format</td>
</tr>
</tbody>
</table>

**Purpose**
Retrieves the length of the label, unit, and format attribute strings associated with the specified dimension.

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

**Description**
The space allocated to hold the label, unit, and format strings must be at least one byte larger than the actual length of the string, to account for the null termination.

**FORTRAN**

```
integer function dsgdiln(dim, label_len, unit_len, format_len)
  integer dim, label_len, unit_len, format_len
```
DFSDgetdims/dsgdims

intn DFSDgetdims(char *filename, intn *rank, int32 dimsizes[], intn maxrank)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>IN</td>
<td>Name of the HDF file</td>
</tr>
<tr>
<td>rank</td>
<td>OUT</td>
<td>Number of dimensions</td>
</tr>
<tr>
<td>dimsizes</td>
<td>OUT</td>
<td>Buffer for the returned dimensions</td>
</tr>
<tr>
<td>maxrank</td>
<td>IN</td>
<td>Size of the storage buffer dimsizes</td>
</tr>
</tbody>
</table>

**Purpose**
Retrieves the number of dimensions \( (rank) \) of the dataset and the sizes of the dimensions \( (dimsizes) \) for the next scientific dataset in the file.

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

**Description**
The maxrank parameter tells DFSDgetdims the size of the array that is allocated for storing the dimsizes array. The value of rank must not exceed the value of maxrank.

The allocation of a buffer for the scientific dataset data should correspond to the values retrieved by DFSDgetdims. The first value in the array dimsizes should equal the first dimension of the array that is allocated to hold the dataset; the second value in dimsizes should equal the second dimension of the dataset, and so forth.

**FORTRAN**

```fortran
integer function dsgdims(filename, rank, dimsizes, maxrank)

character(*) filename
integer rank, maxrank
integer dimsizes(*)
```
DFSDgetdimscale/dsgdisc

intn DFSDgetdimscale(intn dim, int32 size, VOIDP scale)

- **dim**: IN: Dimension this scale corresponds to
- **size**: IN: Size of the scale buffer
- **scale**: OUT: Array of values defining reference points along a specified dimension

**Purpose**: Gets the scale corresponding to the specified dimension.

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

**Description**: The DFSD interface requires the dimension scales to be of the same data type as the corresponding data. To store dimension scales of a different data type than the corresponding data, use the multfile SD interface.

**FORTRAN**

```fortran
integer function dsgdisc(dim, size, scale)

integer dim, size
integer scale(*)
```
**DFSDgetdimstrs/dsgdist**

```fortran
intn DFSDgetdimstrs(intn dim, char *label, char *unit, char *format)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dim</code></td>
<td>IN: Dimension this label, unit and format refer to</td>
</tr>
<tr>
<td><code>label</code></td>
<td>OUT: Label that describes this dimension</td>
</tr>
<tr>
<td><code>unit</code></td>
<td>OUT: Unit to be used with this dimension</td>
</tr>
<tr>
<td><code>format</code></td>
<td>OUT: Format to be used in displaying scale for this dimension</td>
</tr>
</tbody>
</table>

**Purpose**

Retrieves the label, unit, and format attribute strings corresponding to the specified dimension.

**Return value**

Returns **SUCCEED** (or 0) if successful and **FAIL** (or -1) otherwise.

**Description**

The space allocated for the label, unit, and format string must be at least one byte larger than the length of the string to accommodate the null termination. If the length is unknown when the program is written, declare the array size as `1+maxlen_label, maxlen_unit, or maxlen_format` after they are set by `DFSDsetlengths`. The maximum default string length is 255.

**FORTRAN**

```fortran
integer function dsgdist(dim, label, unit, format)

integer dim

character*(*) label, unit, format
```
DFSDgetfillvalue/dsgfill

intn DFSDgetfillvalue(VOIDP fill_value)

  fill_value      OUT:    Fill value

Purpose       Retrieves the fill value of a DFSD scientific dataset.
Return value   Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description    The fill value is set by DFSDsetfillvalue and returned in the variable fill_value. Note that DFSDgetfillvalue does not take a file name as an argument. As a result, a DFSD call to initialize the file information structures is required before calling DFSDgetfillvalue. One such call is DFSDgetdims.

FORTRAN       integer function dsgfill(fill_value)

                      character*(*) fill_value
DFSDgetNT/dsgnt

intn DFSDgetNT(int32 *data_type)

data_type OUT: Data type of data in the scientific dataset

Purpose
Retrieves the data type of the next dataset to be read.

Return value
Returns succeed (or 0) if successful and fail (or -1) otherwise.

Description
Note that DFSDgetNT does not take a file name as an argument. As a result, a DFSD call to initialize the file information structures is required before calling DFSDgetNT. One such call is DFSDgetdims.

Valid values for data_type are of the general form DFNT_. The following are valid symbolic names and their data types:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Symbolic Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit float</td>
<td>DFNT_FLOAT32</td>
<td>5</td>
</tr>
<tr>
<td>64-bit float</td>
<td>DFNT_FLOAT64</td>
<td>6</td>
</tr>
<tr>
<td>8-bit signed int</td>
<td>DFNT_INT8</td>
<td>20</td>
</tr>
<tr>
<td>8-bit unsigned int</td>
<td>DFNT_UINT8</td>
<td>21</td>
</tr>
<tr>
<td>16-bit signed int</td>
<td>DFNT_INT16</td>
<td>22</td>
</tr>
<tr>
<td>16-bit unsigned int</td>
<td>DFNT_UINT16</td>
<td>23</td>
</tr>
<tr>
<td>32-bit signed int</td>
<td>DFNT_INT32</td>
<td>24</td>
</tr>
<tr>
<td>32-bit unsigned int</td>
<td>DFNT_UINT32</td>
<td>25</td>
</tr>
<tr>
<td>8-bit character</td>
<td>DFNT_CHAR8</td>
<td>4</td>
</tr>
</tbody>
</table>

FORTRAN
integer function dsgnt(num_type)

integer num_type
DFSDgetrange/dsgrang

intn DFSDgetrange(VOIDP max, VOIDP min)

  max  OUT:  Maximum value stored with the scientific dataset
  min  OUT:  Maximum value stored with the scientific dataset

Purpose
Retrieves the maximum and minimum values stored with the scientific dataset.

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

Description
The max and min values are set via a call to DFSDsetrange. They are not automatically stored when a dataset is written to a file. The data type of these values is the data type of the dataset array. One implication of this is that in the C version of DFSDgetrange the arguments are pointers, rather than simple variables, whereas in the FORTRAN-77 version they are simple variables of the same type as the data array.

Neither DFSDgetrange nor DFSDgetdata compare the max and min values stored with the dataset to the actual values in the dataset; they merely retrieve the data. As a result, the maximum and minimum values may not always reflect the actual maximum and minimum values in the dataset. In some cases the max and min values may actually lie outside the range of values in the dataset.

FORTRAN
integer function dsgrang(max, min)

character*(*) max, min
DFSDgetslice/dsgslc

DFSDgetslice/dsgslc

intn DFSDgetslice(char *filename, int32 winst[], int32 windims[], VOIDP data, int32 dims[])

filename   IN: Name of HDF file
winst  IN: Array containing the coordinates for the start of the slice
windim  IN: Array containing the dimensions of the slice
data    OUT: Array for returning slice
dims    OUT: Dimensions of array data

Purpose       Reads part of a scientific dataset from a file.
Return value  Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description   DFSDgetslice accesses the dataset last accessed by DFSDgetdims. If DFSDgetdims has not been called for the named file, DFSDgetslice gets a slice from the next dataset in the file. Array winst specifies the coordinates of the start of the slice. Array windims gives the size of the slice. The number of elements in winst and windims must be equal to the rank of the dataset. For example, if the file contains a three-dimensional dataset, winst may contain the values {2, 4, 3}, while windims contains the values {3, 1, 4} and the dims should be at least {3, 1, 4}, the same size as the slice. This will extract a 3 x 4, two-dimensional slice, containing the elements between (2, 4, 3) and (4, 4, 6) from the original dataset.

The data array is the array into which the slice is read. It must be at least as big as the desired slice. The dims array is the array containing the actual dimensions of the array data. The user assigns values to dims before calling DFSDgetslice.

All parameters assume FORTRAN-77-style one-based arrays.

DFSDgetslice is obsolete in favor of DFSDreadslab. DFSDreadslab is the recommended function call to use when reading hyperslabs (previously known as data slices). HDF will continue to support DFSDgetslice only to maintain backward compatibility with HDF applications built on earlier versions of the library.

FORTRAN

integer function dsgslc(filename, winst, windims, data, dims)

character*(*) filename, data
integer winst(*), windims(*), dims(*)
DFSDlastref/dslref

intn DFSDlastref( )

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Retrieves the most recent reference number used in writing or reading a scientific dataset.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return value</td>
<td>Returns the reference number for the last accessed scientific dataset if successful and FAIL (or -1) otherwise.</td>
</tr>
<tr>
<td>Description</td>
<td>DFSDlastref returns the value of the last reference number of a scientific dataset read from or written to the file.</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>integer function dslref( )</td>
</tr>
</tbody>
</table>
**DFSDndatasets/dsnum**

**Purpose**
Returns the number of scientific datasets in the file.

**Return value**
Returns the number of datasets if successful and **FAIL** (or -1) otherwise.

**Description**
In HDF version 3.3, **DFSDndatasets** replaced **DFSDnumber**. In order to maintain backward compatibility with existing HDF applications, HDF will continue to support **DFSDnumber**. However, it is recommended that all new applications use **DFSDndatasets** instead of **DFSDnumber**.

**FORTRAN**

```fortran
integer function dsnum(filename)
    character*(*) filename
```

**filename**
IN: Name of the HDF file
DFSDpre32sdg/dsp32sd

```c
intn DFSDpre32sdg(char *filename, uint16 ref, intn *ispre32)
```

- **filename**  
  IN: The name of the HDF file containing the scientific dataset
- **ref**  
  IN: Reference number of SDG
- **ispre32**  
  OUT: Pointer to results of the pre-HDF version 3.2 inquiry

**Purpose**  
Tests if the scientific dataset with the specified reference number was created by an HDF library earlier than version 3.2.

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

**Description**  
If the scientific dataset was created with a version of HDF prior to version 3.2, `ispre32` will be set to 1, otherwise it will be set to 0. Based on this information, programmers can decide whether or not to transpose the corresponding array.

**FORTRAN**  
```fortran
integer function dsp32sd(filename, ref, ispre32)
character*(*) filename
integer ref, ispre32
```
DFSDputdata/dspdata

intn DFSDputdata(char *filename, intn rank, int32 dimsizes[], VOIDP data)

filename IN: Name of the HDF file
rank IN: Number of dimensions of data array to be stored
dimsizes IN: Buffer for the dimension sizes
data IN: Buffer for the data to be stored

Purpose Writes a scientific data and related information to an HDF file.
Return value Returns SUCCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description DFSDputdata will write data to an existing file by destroying the contents of the original file. Use it with caution. If a new filename is used, DFSDputdata functions exactly like DFSDadddata.

FORTRAN integer function dspdata(filename, rank, dimsizes, data)

character*(*) filename
<valid numeric data type> data
integer rank
integer dimsizes(*)
intn DFSDputslice(int32 windims[], VOIDP source, int32 dims[])

windims  IN: Window dimensions specifying the size of the slice to be written
source   IN: Buffer for the slice
dims     IN: Dimensions of the source array

Purpose  Writes part of a scientific dataset to a file.

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

Description DFSDputslice reads a subset of an array in memory and stores it as part of the
scientific dataset array last specified by DFSDsetdims. Slices must be stored
contiguously.

Array windims ("window dimensions") specifies the size of the slice to be
written. The windims array must contain as many elements as there are
dimensions in the entire scientific dataset array. The source argument is an
array in memory containing the slice and dims is an array containing the
dimensions of the array source.

Notice that windims and dims need not be the same. The windims argument
could refer to a sub-array of source, in which case only a portion of source is
written to the scientific data array.

All parameters assume FORTRAN-77-style one-based arrays.

DFSDputslice is obsolete in favor of DFSDwriteslab. DFSDwriteslab is the
recommended function call to use when writing hyperslabs (previously known
as data slices). HDF will continue to support DFSDputslice only to maintain
backward compatibility with earlier versions of the library.
intn DFSDreadref(char *filename, uint16 ref)

filename IN: Name of the HDF file
ref IN: Reference number for next DFSDgetdata call

Purpose Specifies the reference number for the dataset to be read during the next read operation.

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

Description This routine is commonly used in conjunction with DFANgetlablist, which returns a list of labels for a given tag together with their reference numbers. It provides a sort of random access to scientific datasets.

There is no guarantee that reference numbers appear in sequence in an HDF file, so it is not generally safe to assume that a reference number is an index number of a scientific dataset.

FORTRAN

integer function dsrref(filename, ref)

character(*) filename

integer ref
DFSDreadslab/dsrslab

intn DFSDreadslab(char *filename, int32 start[], int32 slab_size[], int32 stride[], VOIDP buffer, int32 buffer_size[])

filename  IN:  Name of the HDF file
start     IN:  Buffer of size rank containing the coordinates for the start of the slab
slab_size IN:  Buffer of size rank containing the size of each dimension in the slab
stride    IN:  Subsampling (not yet implemented)
buffer    OUT:  Buffer for the returned slab
buffer_size OUT:  Dimensions of the buffer parameter

Purpose  Reads a slab of data from any scientific dataset.
Return value  Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description  DFSDreadslab will access to the scientific dataset following the current one if DFSDgetdims or DFSDgetdata are not called earlier. The start array indices are one-based. The rank of start must be the same as the number of dimensions of the specified variable. The elements of slab_size must be no larger than the dimensions of the scientific dataset in order. The stride feature is not currently implemented. For now just pass the start array as the argument for stride where it will be ignored.

To extract a slab of lower dimension than that of the dataset, enter 1 in the slab_size array for each omitted dimension. For example, to extract a two-dimensional slab from a three-dimensional dataset, specify the beginning coordinates in three dimensions and enter a 1 for the missing dimension in the slab_size array. More specifically, to extract a $3 \times 4$ slab containing the elements $(6, 7, 8)$ through $(8, 7, 11)$ specify the beginning coordinates as $(6, 7, 8)$ and the slab size as $(3, 1, 4)$.

FORTRAN  integer function dsrslab(filename, start, slab_size, stride,  
buffer, buffersize)  

character(*) filename, buffer  
integer start(*), slab_size(*),  
integer stride(*), buffer_size(*)
DFSDrestart/dsfirst

Purpose
Causes the next read command to be read from the first scientific dataset in the file, rather than the scientific dataset following the one that was most recently read.

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

FORTRAN
integer function dsfirst( )
**DFSDsetcal/dsscal**

```c
intn DFSDsetcal(float64 cal, float64 cal_err, float64 offset, float64 offset_err, int32 data_type)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cal</code></td>
<td>Calibration factor</td>
</tr>
<tr>
<td><code>cal_err</code></td>
<td>Calibration error</td>
</tr>
<tr>
<td><code>offset</code></td>
<td>Uncalibrated offset</td>
</tr>
<tr>
<td><code>offset_err</code></td>
<td>Uncalibrated offset error</td>
</tr>
<tr>
<td><code>data_type</code></td>
<td>Data type of uncalibrated data</td>
</tr>
</tbody>
</table>

**Purpose**
Sets the calibration information associated with data

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

**Description**
This routine sets the calibration record associated with a dataset. A calibration record contains four 64-bit floating point values followed by a 32-bit integer, to be interpreted as follows:

- `cal` - calibration factor
- `cal_err` - calibration error
- `offset` - uncalibrated offset
- `offset_err` - uncalibrated offset error
- `data_type` - data type of uncalibrated data

The relationship between a value `y` stored in a dataset and the actual value `yi` is defined as:

\[ y = \text{cal} \times (yi - \text{offset}) \]

The variable `offset_err` contains a potential error of `offset`, and `cal_err` contains a potential error of `cal`. Currently the calibration record is provided for information only. The SD interface performs no operations on the data based on the calibration tag.

`DFSDsetcal` works like other `DFSDset*` routines, with one exception: the calibration information is automatically cleared after a call to `DFSDputdata` or `DFSDadddata`. Hence, `DFSDsetcal` must be called again for each dataset that is to be written.

As an example, suppose the values in a dataset `y[]` are as follows:

\[ y[6]=\{1001.0, 1002.0, 1002.5, 1005.5, 1013.0, 1040.5\} \]

By defining \( \text{cal} = 0.50 \) and \( \text{offset} = -200.0 \) and applying the calibration formula, the calibrated dataset `iy[]` becomes as follows:

\[ iy[6]=\{2, 4, 5, 11, 26, 81\} \]

The array `iy[]` can then be stored as integers.
FORTRAN

integer function dsscal(cal, cal_err, offset, offset_err, data_type)

real*8 cal, cal_err, offset, offset_err

integer data_type
DFSDsetdatastrs/dssdast

intn DFSDsetdatastrs(char *label, char *unit, char *format, char *coordsys)

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>IN: Label describing the data</td>
</tr>
<tr>
<td>unit</td>
<td>IN: Unit to be used with the data</td>
</tr>
<tr>
<td>format</td>
<td>IN: Format to be used in displaying the data</td>
</tr>
<tr>
<td>coordsys</td>
<td>IN: Coordinate system of the data</td>
</tr>
</tbody>
</table>

**Purpose**
Sets the label, unit, format, and coordinate system for the next dataset written to file.

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

**FORTRAN**

```
integer function dssdast(label, unit, format, coordsys)
character*(*) label, unit, format, coordsys
```
DFSDsetdims/dssdims

DFSDsetdims (intn rank, int32 dimsizes[])

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rank</td>
<td>Number of dimensions</td>
</tr>
<tr>
<td>dimsizes</td>
<td>Dimensions of the scientific dataset</td>
</tr>
</tbody>
</table>

**Purpose**
Sets the rank and dimension sizes for all subsequent scientific datasets written to the file.

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

**Description**
This routine must be called before calling either DFSDsetdimstrs or DFSDsetdimscale. DFSDsetdims need not be called if other set routines are not called and the correct dimensions are supplied in DFSDputdata or DFSDadddata.

If the rank or dimension sizes change, all previous set calls are cleared, except for the data type, which is set by calling DFSDsetNT.

**FORTRAN**

```
integer function dssdims(rank, dimsizes)
    integer rank
    integer dimsizes(*)
```

2-324  December 30, 2003
DFSDsetdimscale/dssdisc

```c
intn DFSDsetdimscale (intn dim, int32 dimsize, VOIDP scale)
```

- **dim**  
  IN: Dimension this scale corresponds to

- **dimsize**  
  IN: Size of the *scale* buffer

- **scale**  
  IN: Buffer for the scale values

**Purpose**  
Defines the scale for a dimension.

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

**Description**  
A scale is a one-dimensional array whose values describe reference points along one dimension of the dataset. For example, a two-dimensional dataset representing points on a map could have two scales, one representing points of latitude, and the other points of longitude.

**FORTRAN**

```fortran
integer function dssdisc (dim, dimsize, scale)

integer dim

integer dimsize(*), scale(*)
```
DFSDsetdimstrs/dssdist

```c
intn DFSDsetdimstrs(intn dim, char *label, char *unit, char *format)
```

- **dim** IN: Dimension this label, unit and format refer to
- **label** IN: Label that describes this dimension
- **unit** IN: Unit to be used with this dimension
- **format** IN: Format to be used to display scale

**Purpose**
Sets the label, unit, and format strings corresponding to the specified dimension.

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

**Description**
In both FORTRAN-77 and C programs, `dim = 1` for the first dimension, and `dim = 2` for the second dimension. If the user is not interested in one or more strings, empty strings can be used as parameters for the `DFSDsetdimstrs` call. For example, `DFSDsetdimstrs(1, “vertical”, “”, “”)` will set the label for the first dimension to “vertical” and set the unit and format to empty strings.

**FORTRAN**
```fortran
integer function dssdist(dim, label, unit, format)

integer dim

character*(*) label, unit, format
```
DFSDsetfillvalue/dssfill

intn DFSDsetfillvalue(VOIDP fill_value)

fill_value IN: Fill value

Purpose Set the value used to fill in any unwritten location in a scientific dataset.

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

Description It is assumed that the fill value has the same data type as the dataset. Once the fill value is set for a particular SDS, it cannot be changed. If DFSDsetfillvalue is called before the first call to DFSDstarts lab, DFSDstarts lab will set the fill value tag attribute to the value specified in the DFSDsetfillvalue call, but will not actually write out the fill value when DFSDwriteslab is called. However, if DFSDsetfillvalue is called after the first call the DFSDstarts lab, the fill value tag attribute will be set by DFSDsetfillvalue and the fill value will be written to the slab during the DFSDwriteslab call.

FORTRAN

integer function dssfill(fill_value)

character(*) fill_value
DFSDsetlengths/dsslens

```fortran
intn DFSDsetlengths(intn label_len, intn unit_len, intn format_len, intn coords_len)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>label_len</code></td>
<td>Maximum length of label strings</td>
</tr>
<tr>
<td><code>unit_len</code></td>
<td>Maximum length of unit strings</td>
</tr>
<tr>
<td><code>format_len</code></td>
<td>Maximum length of format strings</td>
</tr>
<tr>
<td><code>coords_len</code></td>
<td>Maximum length of coordinate system strings</td>
</tr>
</tbody>
</table>

**Purpose**: Sets the maximum lengths for the strings that will hold labels, units, formats, and the name of the coordinate system.

**Return value**: Returns `SUCCEED` (or 0) if successful and `FAIL` (or -1) otherwise.

**Description**: The lengths set by this routine are used by the routines `DFSDgetdimstrs` and `DFSDgetdatastrs` to determine the maximum lengths of strings that they get from the file.

Normally, `DFSDsetlengths` is not needed. If it is not called, default maximum lengths of 255 are used for all strings.

```fortran
integer function dsslens(label_len, unit_len, format_len, coords_len)

integer label_len, unit_len, format_len, coords_len
```
**DFSDKsetNT/dssnt**

```c
intn DFSDKsetNT(int32 data_type)
```

*data_type*  
**IN:** Data type

**Purpose**  
Sets the data type of the data to be written in the next write operation.

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

**Description**  
DFSDKsetNT must be called if a data type other than float32 is to be stored. DFSDKsetNT and DFSDKsetdims can be called in any order, but they should be called before any other DFSDKset* functions and before DFSDKputdata or DFSDKdaddata.

The following symbolic names can be used as the value of *data_type*:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Symbolic Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit float</td>
<td>DFNT_FLOAT32</td>
<td>5</td>
</tr>
<tr>
<td>64-bit float</td>
<td>DFNT_FLOAT64</td>
<td>6</td>
</tr>
<tr>
<td>8-bit signed int</td>
<td>DFNT_INT8</td>
<td>20</td>
</tr>
<tr>
<td>8-bit unsigned int</td>
<td>DFNT_UINT8</td>
<td>21</td>
</tr>
<tr>
<td>16-bit signed int</td>
<td>DFNT_INT16</td>
<td>22</td>
</tr>
<tr>
<td>16-bit unsigned int</td>
<td>DFNT_UINT16</td>
<td>23</td>
</tr>
<tr>
<td>32-bit signed int</td>
<td>DFNT_INT32</td>
<td>24</td>
</tr>
<tr>
<td>32-bit unsigned int</td>
<td>DFNT_UINT32</td>
<td>25</td>
</tr>
<tr>
<td>8-bit character</td>
<td>DFNT_CHAR8</td>
<td>4</td>
</tr>
</tbody>
</table>

**FORTRAN**

```fortran
integer function dssnt(num_type)
```

```fortran
integer num_type
```
DFSDsetrange/dssrang

DFSDsetrange/dssrang

intn DFSDsetrange(VOIDP max, VOIDP min)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>max</td>
<td>IN: Highest value in the range</td>
</tr>
<tr>
<td>min</td>
<td>IN: Lowest value in the range</td>
</tr>
</tbody>
</table>

**Purpose**
Stores the specified maximum and minimum data values.

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

**Description**
It is assumed that the data type of `max` and `min` is the same as the type of the data. One implication of this is that in the C version of DFSDsetrange the arguments are pointers, rather than simple variables, whereas in the FORTRAN-77 version they are simple variables of the same type as the data array.

This routine does not compute the maximum and minimum values; it merely stores the values it is given. As a result, the maximum and minimum values may not always reflect the actual maximum and minimum values in the data array.

When the maximum and minimum values are written to a file, the HDF element that holds these values is cleared, because it is assumed that subsequent datasets will have different values for max and min.

**FORTRAN**

```fortran
integer function dssrang(max, min)
character*(*) max, min
```
DFSDstartslab/dssslab

intn DFSDstartslab(char *filename)

filename IN: Name of the HDF file

Purpose Prepares the DFSD interface to write a slab of data to a scientific dataset.

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

Description DFSDsetdims must be called before calling DFSDstartslab. No call which involves a file open may be made after a DFSDstartslab call until DFSDendslab is called. This routine will write out the fill values if DFSDsetfillvalue is called before this routine.

FORTRAN integer function dssslab(filename)

character*(*) filename
intn DFSDstartslicefilename)

filename IN: Name of the HDF file

**Purpose**
Prepares the interface to write a data slice to the specified file.

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

**Description**
Before calling **DFSDstartslicefilename** must be called to specify the dimensions of the dataset to be written to the file. **DFSDstartslicefilename** always appends a new dataset to an existing file.

Also, **DFSDstartslicefilename** must be called before **DFSDputslicefilename** or **DFSDendslicefilename**.

**DFSDstartslicefilename** is obsolete in favor of **DFSDstartslabfilename**. **DFSDstartslabfilename** is the recommended function call to use when beginning hyperslab operations. HDF will continue to support **DFSDstartslicefilename** only to maintain backward compatibility earlier versions of the library.

**FORTRAN**

integer function dssslc(filename)

character*(*)(filename)
DFSDwriteref/dswref

intn DFSDwriteref(char *filename, uint16 ref)

filename IN: Name of the HDF file
ref IN: Reference number for next add or put operation

Purpose
Specifies the reference number, ref, of the dataset to be overwritten next by DFSDputdata or DFSDadddata.

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

Description
DFSDwriteref verifies the reference number's existence before returning. If a non-existent reference number is specified, an error code will be returned.

As this routine alters data in a destructive manner, DFSDwriteref should be used with caution.

FORTRAN
integer function dswref(filename, ref)

character(*) filename
integer ref
DFSDwriteslab/dswslab

`intn DFSDwriteslab(int32 start[], int32 stride[], int32 count[], VOIDP data)`

`start` **IN:** Array containing the starting coordinates of the slab

`stride` **IN:** Array containing the dimensions for subsampling

`count` **IN:** Array containing the size of the slab

`data` **IN:** Array to hold the floating point data to be written

**Purpose**

Writes a slab of data to a scientific dataset.

**Return value**

Returns `SUCCEED` (or 0) if successful and `FAIL` (or -1) otherwise.

**Description**

The `start` indices are relative to 1. The rank of `start` must be the same as the number of dimensions of the specified variable. The elements of `start` must be no larger than the scientific dataset's dimensions in order. The stride feature is not currently implemented. For now just pass the `start` array as the argument for the `stride` parameter, where it will be ignored.

The rank of `count` must be the same as the number of dimensions of the specified variable. The elements of `count` must be no larger than the scientific dataset's dimensions in order. The order in which the data will be written into the specified hyperslab is with the last dimension varying fastest. The data should be of the appropriate type for the dataset. Note that neither the compiler nor HDF software can detect if the wrong type of data is used.

**FORTRAN**

```fortran
integer function dswslab(start, stride, count, data)

integer start(*), stride(*), count(*)

character*(*) data
```
# Happendable

**intn Happendable(int32 h_id)**

- **$h_id$**  
  **IN:** Access identifier returned by **Hstartwrite**

**Purpose**  
Specifies that the specified element can be appended to

**Return value**  
Returns **SUCCEED** (or 0) if data element can be appended and **FAIL** (or -1) otherwise.

**Description**  
If a data element is at the end of a file **Happendable** allows **Hwrite** to append data to it, converting it to linked-block element only when necessary.
Hcache

intn Hcache(int32 file_id, intn cache_switch)

file_id IN: File identifier returned by Hopen
cache_switch IN: Flag to enable or disable caching

Purpose Enables low-level caching for the specified file.
Return value Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description If file_id is set to CACHE_ALL_FILES, then the value of cache_switch is used to modify the default file cache setting.

Valid values for cache_switch are: TRUE (or 1) to enable caching and FALSE (or 0) to disable caching.
Hdeldd

intn Hdeldd(int32 file_id, uint16 tag, uint16 ref)

file_id IN: File identifier returned by Hopen

tag IN: Tag of data descriptor to be deleted

ref IN: Reference number of data descriptor to be deleted

Purpose
Deletes a tag and reference number from the data descriptor list.

Return value
Returns succeed (or 0) if successful and fail (or -1) otherwise.

Description
Once the data descriptor is removed, the data in the data object becomes inaccessible and is marked as such. To remove inaccessible data from an HDF file, use the utility hdfpack.

Hdeldd only deletes the specified tag and reference number from the data descriptor list. Data objects containing the deleted tag and reference number are not automatically updated. For example, if the tag and reference number deleted from the descriptor list referenced an object in a vgroup, the tag and reference number will still exist in the vgroup even though the data is inaccessible.
Hendaccess

intn Hendaccess(int32 h_id)

h_id(IN): Access identifier returned by Hstartread, Hstartwrite, or Hnextread

Purpose: Terminates access to a data object by disposing of the access identifier.

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

Description: The number of active access identifiers is limited to MAX_ACC as defined in the hlimits.h header file. Because of this restriction, it is very important to call Hendaccess immediately following the last operation on a data element.

When developing new interfaces, a common mistake is to omit calling Hendaccess for all of the elements accessed. When this happens, Hclose will return FAIL, and a dump of the error stack will report the number of active access identifiers. Refer to the Reference Manual page on HEprint.

This is a difficult problem to debug because the low levels of the HDF library cannot determine who and where an access identifier was originated. As a result, there is no automated method of determining which access identifiers have yet to be released.
Hendbitaccess

intn Hendbitaccess(int32 h_id, intn flushbit)

h_id  IN: Identifier of the bit-access element to be disposed of
flushbit  IN: Specifies how the leftover bits are to be flushed

Purpose  Disposes of the specified bit-access file element.
Return value  Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description  If called after a bit-write operation, Hendbitaccess flushes all buffered bits to the dataset, then calls Hendaccess.

“Leftover bits" are bits that have been buffered, but are fewer than the number of bits defined by BITNUM, which is usually set to 8.

Valid codes for flushbit are: 0 for flush with zeros, 1 for flush with ones and -1 for dispose of leftover bits
Hexist

intn Hexist(int32 h_id, uint16 search_tag, uint16 search_ref)

h_id IN:  Access identifier returned by Hstartread, Hstartwrite, or Hnextread

search_tag IN:  Tag of the object to be searched for

search_ref IN:  Reference number of the object to be searched for

Purpose  Locates an object in an HDF file.

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

Description  Simple interface to Hfind that determines if a given tag/reference number pair exists in a file. Wildcards apply.

Hfind performs all validity checking; this is just a very simple wrapper around it.
## Hfidinquire

void Hfidinquire(int32 file_id, char *filename, intn *access, intn *attach)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file_id</td>
<td>IN</td>
<td>File identifier returned by Hopen</td>
</tr>
<tr>
<td>filename</td>
<td>OUT</td>
<td>Complete path and filename for the file</td>
</tr>
<tr>
<td>access</td>
<td>OUT</td>
<td>Access mode file is opened with</td>
</tr>
<tr>
<td>attach</td>
<td>OUT</td>
<td>Number of access identifiers attached to the file</td>
</tr>
</tbody>
</table>

### Purpose

Returns file information through a reference of its file identifier.

### Return value

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

### Description

Gets the complete path name, access mode, and number of access identifiers associated with a file. The filename parameter is a pointer to a character pointer which will be modified when the function returns. Upon completion, filename is set to point to the file name in internal storage. All output parameters must be non-null pointers.
Hfind

int Hfind(int32 file_id, uint16 search_tag, uint16 search_ref, uint16 *find_tag, uint16 *find_ref, int32 *find_offset, int32 *find_length, intn direction)

file_id IN: File identifier returned by Hopen
search_tag IN: The tag to search for or DFTAG_WILDCARD
search_ref IN: Reference number to search for or DFREF_WILDCARD
find_tag IN/OUT: If (*find_tag == 0) and (*find_ref == 0) then start the search from either the beginning or the end of the file. If the object is found, the tags of the object will be returned here.
find_ref IN/OUT: If (*find_tag == 0) and (*find_ref == 0) then start the search from either the beginning or the end of the file. If the object is found, the reference numbers of the object will be returned here.
find_offset OUT: Offset of the data element found
find_length OUT: Length of the data element found
direction IN: Direction to search in DF_FORWARD searches forward from the current location, and DF_BACKWARD searches backward from the current location

Purpose Locates the next object to be searched for in an HDF file.
Return value Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description Hfind searches for the next data element that matches the specified tag and reference number. Wildcards apply. If direction is DF_FORWARD, searching is forward from the current position in the file, otherwise DF_BACKWARD specifies backward searches from the current position in the file.

If find_tag and find_ref are both set to 0, this indicates the beginning of a search, and the search will start from the beginning of the file if the direction is DF_FORWARD and from the end of the file if the direction is DF_BACKWARD.
**Hgetbit**

```c
int Hgetbit(int32 h_id)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>h_id</code></td>
<td>Bit-access element identifier</td>
</tr>
</tbody>
</table>

**Purpose**
Reads one bit from the specified bit-access element.

**Return value**
Returns the bit read (or 0 or 1) if successful and FAIL (or -1) otherwise.

**Description**
This function is a wrapper for *Hbitread*. 
### Hgetelement

**Signature**

```c
int32 Hgetelement(int32 file_id, uint16 tag, uint16 ref, uint8 *data)
```

**Parameters**

- `file_id`<br>IN: File identifier returned by `Hopen`
- `tag`<br>IN: Tag of the data element to be read
- `ref`<br>IN: Reference number of the data element to be read
- `data`<br>OUT: Buffer the element will be read into

**Purpose**

Reads the data element for the specified tag and reference number and writes it to the `data` buffer.

**Return value**

Returns the number of bytes read if successful and `FAIL` (or `-1`) otherwise.

**Description**

It is assumed that the space allocated for the buffer is large enough to hold the data.
Hinquire

intn Hinquire(int32 h_id, int32 *file_id, uint16 *tag, uint16 *ref, int32 *length, int32 *offset, int32 *position, int16 *access, int16 *special)

- **h_id** (IN): Access identifier returned by Hstartread, Hstartwrite, or Hnextread
- **file_id** (OUT): File identifier returned by Hopen
- **tag** (OUT): Tag of the element pointed to
- **ref** (OUT): Reference number of the element pointed to
- **length** (OUT): Length of the element pointed to
- **offset** (OUT): Offset of the element in the file
- **position** (OUT): Current position within the data element
- **access** (OUT): The access type for this data element
- **special** (OUT): Special code

**Purpose**: Returns access information about a data element.

**Return value**: Returns SUCCEED (or 0) if the access identifier points to a valid data element and FAIL (or -1) otherwise.

**Description**: If h_id is a valid access identifier the access type (read or write) is set regardless of whether or not the return value is FAIL (or -1). If h_id is invalid, the function returns FAIL (or -1) and the access type is set to zero. To avoid excess information, pass NULL for any unnecessary pointer.
Hlength

int32 Hlength(int32 file_id, uint16 tag, uint16 ref)

file_id: IN: File identifier returned by Hopen

tag: IN: Tag of the data element

ref: IN: Reference number of the data element

Purpose
Returns the length of a data object specified by the tag and reference number.

Return value
Returns the length of data element if found and FAIL (or -1) otherwise.

Description
Hlength calls Hstartread, HQuerylength, and Hendaccess to determine the length of a data element. Hlength uses Hstartread to obtain an access identifier for the specified data object.

Hlength will return the correct data length for linked-block elements, however it is important to remember that the data in linked-block elements is not stored contiguously.
Hnewref

```c
uint16 Hnewref(int32 file_id)
```

**file_id**

IN: File identifier returned by **Hopen**

**Purpose**

Returns a reference number that can be used with any tag to produce a unique tag/reference number pair.

**Return value**

Returns the reference number if successful and 0 otherwise.

**Description**

Successive calls to **Hnewref** will generate reference number values that increase by one each time until the highest possible reference number has been returned. At this point, additional calls to **Hnewref** will return an increasing sequence of unused reference number values starting from 1.
Hnextread

intn Hnextread(int32 $h_id$, uint16 $tag$, uint16 $ref$, int $origin$)

$h_id$ IN: Access identifier returned by Hstartread or previous Hnextread
$tag$ IN: Tag to search for
$ref$ IN: Reference number to search for
$origin$ IN: Position to begin search: DF_START or DF_CURRENT

Purpose
Searches for the next data descriptor that matches the specified tag and reference number.

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

Description
Wildcards apply. If origin is DF_START, the search will start at the beginning of the data descriptor list. If origin is DF_CURRENT, the search will begin at the current position. Searching backwards from the end of a data descriptor list is not yet implemented.

If the search is successful, the access identifier reflects the new data element, otherwise it is not modified.
**Hnumber/hnumber**

\[ \text{int32 Hnumber(int32 file_id, uint16 tag)} \]

- **file_id** IN: File identifier returned by **Hopen**
- **tag** IN: Tag to be counted

**Purpose**
Returns the number of instances of a tag in a file.

**Return value**
Returns the number of instances of a tag in a file if successful, and **FAIL** (or -1) otherwise.

**Description**
**Hnumber** determines how many objects with the specified tag are in a file. To determine the total number of objects in a file, set the \( \text{tag} \) argument to **DFTAG_WILDCARD**. Note that a return value of zero is not a fail condition.

**FORTRAN**
\[ \text{integer function hnumber(file_id, tag)} \]
\[ \text{integer file_id, tag} \]
Hoffset

```
hoffset(int32 file_id, uint16 tag, uint16 ref)
```

**file_id**  
IN: File identifier returned by `Hopen`

**tag**  
IN: Tag of the data element

**ref**  
IN: Reference number of the data element

**Purpose**  
Returns the offset of a data element in the file.

**Return value**  
Returns the offset of the data element if the data element exists and `FAIL` (or `-1`) otherwise.

**Description**  
`Hoffset` calls `Hstartread`, `HQueryoffset`, and `Hendaccess` to determine the length of a data element. `Hoffset` uses `Hstartread` to obtain an access identifier for the specified data object.

`Hoffset` will return the correct offset for a linked-block element, however it is important to remember that the data in linked-block elements is not stored contiguously. The offset returned by `Hoffset` only reflects the position of the first data block.

`Hoffset` should not be used to determine the offset of an external element. In this case, `Hoffset` returns zero, an invalid offset for HDF files.
Hputbit

intn Hputbit(int32 h_id, intn bit)

h_id    IN:    Bit-access element identifier
bit     IN:    Bit to be written

Purpose  Writes one bit to the specified bit-access element.
Return value  Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description  This function is a wrapper for Hbitwrite.
Hputelement

Hputelement

int32 Hputelement(int32 file_id, uint16 tag, uint16 ref, uint8 *data, int32 length)

- **file_id** IN: File identifier returned by [Hopen](#)
- **tag** IN: Tag of the data element to add or replace
- **ref** IN: Reference number of the data element to add or replace
- **data** IN: Pointer to data buffer
- **length** IN: Length of data to write

**Purpose**

Writes a data element or replaces an existing data element in a HDF file.

**Return value**

Returns the number of bytes written if successful and **FAIL** (or -1) otherwise.
**Hread**

```c
int32 Hread(int32 h_id, int32 length, VOIDP data)
```

- **h_id** IN: Access identifier returned by **Hstartread**, **Hstartwrite**, or **Hnextread**
- **length** IN: Length of segment to be read
- **data** OUT: Pointer to the data array to be read

**Purpose**
Reads the next segment in a data element.

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

**Description**
**Hread** begins reading at the current file position, reads the specified number of bytes, and increments the current file position by one. Calling **Hread** with the `length = 0` reads the entire data element. To reposition an access identifier before writing data, use **Hseek**.

If `length` is longer than the data element, the read operation is terminated at the end of the data element, and the number of read bytes is returned. Although only one access identifier is allowed per data element, it is possible to interlace reads from multiple data elements in the same file. It is assumed that data is large enough to hold the specified data length.
Hseek

Hseek(int32 h_id, int32 offset, intn origin)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>h_id</td>
<td>Access identifier returned by Hstartread, Hstartwrite, or Hnextread</td>
</tr>
<tr>
<td>offset</td>
<td>Number of bytes to seek to from the origin</td>
</tr>
<tr>
<td>origin</td>
<td>Position of the offset origin</td>
</tr>
</tbody>
</table>

Purpose
Sets the access pointer to an offset within a data element.

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

Description
Sets the seek position for the next Hread or Hwrite operation by moving an access identifier to the specified position in a data element. The origin and the offset arguments determine the byte location for the access identifier. If origin is set to DF_START, the offset is added to the beginning of the data element. If origin is set to DF_CURRENT, the offset is added to the current position of the access identifier.

Valid values for origin are: DF_START (the beginning of the file) or DF_CURRENT (the current position in the file).

This routine fails if the access identifier if h_id is invalid or if the seek position is outside the range of the data element.
Hsetlength

\[ \text{int32 \ Hsetlength(int32 \ file\_id, \ int32 \ length)} \]

- **file\_id** IN: File identifier returned by `Hopen`
- **length** IN: Length of the new element

**Purpose**
Specifies the length of a new HDF element.

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

**Description**
This function can only be used when called after `Hstartaccess` on a new data element and before any data is written to that element.
Hshutdown

int32 Hshutdown( )

Purpose       Deallocates buffers previously allocated in other H routines.
Return value  Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.
Description   Should only be called by the function HDFend.
Htagnewref

int32 Htagnewref(int32 file_id, uint16 tag)

file_id   IN: Access identifier returned by Hstartread or Hnextread

Tag to be identified with the returned reference number

Purpose

Returns a reference number that is unique for the specified file that will correspond to the specified tag. Creates a new tag/reference number pair.

Return value

Returns the reference number if successful and 0 otherwise.

Description

Successive calls to Hnewref will generate an increasing sequence of reference number values until the highest possible reference number value has been returned. It will then return unused reference number values starting from 1 in increasing order.
Htrunc

int32 Htrunc(int32 h_id, int32 trunc_len)

h_id       IN:    Access identifier returned by Hstartread or Hnextread
trunc_len  IN:    Length to truncate element

Purpose    Truncates the data object specified by the h_id to the length trunc_len.

Return value Returns the length of a data element if found and FAIL (or -1) otherwise.

Description Htrunc does not handle special elements.
Hwrite

int32 Hwrite(int32 h_id, int32 length, VOIDP data)

- **h_id** IN: Access identifier returned by Hstartwrite
- **len** IN: Length of segment to be written
- **data** IN: Pointer to the data to be written

**Purpose**
Writes the next data segment to a specified data element.

**Return value**
Returns the length of the segment actually written if successful and FAIL (or -1) otherwise.

**Description**
Hwrite begins writing at the current position of the access identifier, writes the specified number of bytes, then moves the access identifier to the position immediately following the last accessed byte. Calling Hwrite with length = 0 results in an error condition. To reposition an access identifier before writing data, use Hseek.

If the space allocated in the data element is smaller than the length of data, the data is truncated to the length of the data element. Although only one access identifier is allowed per data element, it is possible to interlace writes to more than one data element in a file.
### HDFclose/hdfclose

```c
intn HDFclose(int32 file_id)
```

**file_id**  
IN: File identifier returned by **Hopen**

**Purpose**  
Closes the access path to the file.

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

**Description**  
The file identifier `file_id` is validated before the file is closed. If the identifier is valid, the function closes the access path to the file.

If there are still access identifiers attached to the file, the error code **DFE_OPENAID** is returned and the file is not closed. This is a common occurrence when developing new interfaces. See **Hendaccess** for further discussion of this problem.

**FORTRAN**  
```fortran
integer function hdfclose(file_id)

integer file_id
```
int32 HDFopen(char *filename, intn access, int16 n_dds)

filename IN: Complete path and filename for the file to be opened
access IN: File access code
n_dds IN: Number of data descriptors in a block if a new file is to be created

Purpose Provides an access path to an HDF file by reading all the data descriptor blocks into memory.

Return value Returns the file identifier if successful and FAIL (or -1) otherwise.

Description If given a new file name, HDFopen will create a new file using the specified access type and number of data descriptors. If given an existing file name, HDFopen will open the file using the specified access type and ignore the n_dds argument.

HDF provides several file access code definitions:
DFACC_READ - Open for read only. If file does not exist, an error condition results.
DFACC_CREATE - If file exists, delete it, then open a new file for read/write.
DFACC_WRITE - Open for read/write. If file does not exist, create it.

If a file is opened and an attempt is made to reopen the file using DFACC_CREATE, HDF will issue the error DFE_ALROPEN. If the file is opened with read only access and an attempt is made to reopen the file for write access using DFACC_RDWR, DFACC_WRITE, or DFACC_ALL, HDF will attempt to reopen the file with read and write permissions.

Upon successful exit, the named file is opened with the relevant permissions, the data descriptors are set up in memory, and the associated file_id is returned. For new files, the appropriate file headers are also set up.

FORTRAN integer function hdfopen(filename, access, n_dds)

character(*) filename
integer access, n_dds
**HEclear**

VOID HEclear( )

<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>Clears all information on reported errors from the error stack.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Return value</strong></td>
<td>None.</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>HEpush creates an error stack. HEclear is then used to clear this stack after any errors are processed.</td>
</tr>
</tbody>
</table>
HEpush

VOID HEpush(int16 error_code, char *funct_name, char *file_name, intn line)

error_code IN: HDF error code corresponding to the error
funct_name IN: Name of function in which the error occurred
file_name IN: Name of file in which the error occurred
line IN: Line number in the file that error occurred

Purpose Pushes a new error onto the error stack.
Return value None.
Description HEpush pushes the file name, function name, line number, and generic
description of the error onto the error stack. HEreport can then be used to give
a more case-specific description of the error.

If the stack is full, the error is ignored. HEpush assumes that the character
strings funct_name and file_name are in semi-permanent storage, so only
pointers to the strings are saved.
**HEreport**

VOID HEreport(char *format, ...)

*format* IN: Output string specification

**Purpose**

Adds a text string to the description of the most-recently-reported error (only one text string per error).

**Return value**

None

**Description**

HEpush places on the error stack the file name, function name, line number, and a generic description of the error type. HEreport can then be used to give a more case-specific description of the error. Only one additional annotation can be attached to each error report.

The format argument must conform to the string specification requirements of printf.
HEvalue

int16 HEvalue(int32 level)

level IN: Level of the error stack to be returned

Purpose Returns an error from the specified level of the error stack.

Return value The error code if successful for DFE_NONE otherwise.
H Functions for Low-level Development

The following H functions are documented in the Function Specifications chapter of the HDF Specification and Developer’s Guide. Note that the functions documented in that chapter include the following:

- Several H-level functions that are intended for low-level development and are not generally appropriate for application programming. These functions are documented only in the HDF Specification and Developer’s Guide.
- Several more-broadly used H-level functions that are also documented in this reference manual.

Opening and closing files

- Hopen
- Hclose

Locating elements for accessing and getting information

- Hstartread
- Hnextread
- Hstartwrite
- Hstartaccess
- Hendaccess
- Hinquire
- Hshdf
- Hnumber
- Hgetlibversion
- Hgetfileversion

Reading and writing entire data elements

- Hputelement
- Hgetelement

Reading and writing part of a data element

- Hread
- Hwrite
- Hseek

Manipulating data descriptors

- Hdupdd
- Hdeldd
- Hnewref

Managing special data elements

- HLcreate
- HXcreate
H Functions for Low-level Development

Data set chunking
- HMCcreate
- HMCwriteChunk
- HMCreadChunk
- HMCsetMaxcache
- HMCPstwrite
- HMCPseek
- HMCPchunkread
- HMCPread
- HMCPchunkwrite
- HMCPwrite
- HMCPcloseAID
- HMCPendaccess
- HMCPinfo
- HMCPinquire

Development routines
- HDgettagname
- HDgetspace
- HDfreespace
- HDstrncpy

Error reporting
- HEprint
- HEclear
- HERROR
- HReport

Other
- Hsync
# HDF Definition List

## 3.1 Definition List Overview

This section of the Reference Manual contains a listing of all definitions used with HDF routines. The definitions are categorized by their name prefix (the portion of the name before the underscore) into tables. The tables themselves are alphabetized by name.

This section is primarily intended to be of use to Fortran programmers whose compilers do not support include files, and need to know the values of the definitions so that they can be explicitly defined in their programs.

### TABLE 3A

*_INTERLACE* - Interlace Mode Codes

<table>
<thead>
<tr>
<th>Definition Name</th>
<th>Definition Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FULL_INTERLACE</td>
<td>0</td>
</tr>
<tr>
<td>NO_INTERLACE</td>
<td>1</td>
</tr>
</tbody>
</table>

### TABLE 3B

*_WILDCARD* - Wildcard Code

<table>
<thead>
<tr>
<th>Definition Name</th>
<th>Definition Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFREF_WILDCARD</td>
<td>0</td>
</tr>
</tbody>
</table>

### TABLE 3C

AN_* - Multifile Annotation Codes

<table>
<thead>
<tr>
<th>Definition Name</th>
<th>Definition Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AN_DATA_LABEL</td>
<td>0</td>
</tr>
<tr>
<td>AN_DATA_DESC</td>
<td>1</td>
</tr>
<tr>
<td>AN_FILE_LABEL</td>
<td>2</td>
</tr>
<tr>
<td>AN_FILE_DESC</td>
<td>3</td>
</tr>
</tbody>
</table>

### TABLE 3D

COMP_* - Raster Image Compression Codes

<table>
<thead>
<tr>
<th>Definition Name</th>
<th>Definition Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMP_NONE</td>
<td>0</td>
</tr>
<tr>
<td>COMP_RLE</td>
<td>11</td>
</tr>
<tr>
<td>COMP_JMCOMP</td>
<td>12</td>
</tr>
<tr>
<td>COMP_JPEG</td>
<td>2</td>
</tr>
</tbody>
</table>
### TABLE 3E
**COMP_CODE_* - General Compression Codes**

<table>
<thead>
<tr>
<th>Definition Name</th>
<th>Definition Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMP_CODE_NONE</td>
<td>0</td>
</tr>
<tr>
<td>COMP_CODE_RLE</td>
<td>1</td>
</tr>
<tr>
<td>COMP_CODE_NBIT</td>
<td>2</td>
</tr>
<tr>
<td>COMP_CODE_SKPHUFF</td>
<td>3</td>
</tr>
<tr>
<td>COMP_CODE_DEFLATE</td>
<td>4</td>
</tr>
<tr>
<td>COMP_CODE_INVALID</td>
<td>5</td>
</tr>
</tbody>
</table>

### TABLE 3F
**DF_* - Maximum Length Codes**

<table>
<thead>
<tr>
<th>Definition Name</th>
<th>Definition Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF_MAXFLEN</td>
<td>256</td>
</tr>
</tbody>
</table>

### TABLE 3G
**DFACC_* - File Access Codes**

<table>
<thead>
<tr>
<th>Definition Name</th>
<th>Definition Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFACC_READ</td>
<td>1</td>
</tr>
<tr>
<td>DFACC_WRITE</td>
<td>2</td>
</tr>
<tr>
<td>DFACC_CREATE</td>
<td>4</td>
</tr>
<tr>
<td>DFACC_ALL</td>
<td>7</td>
</tr>
<tr>
<td>DFACC_RDONLY</td>
<td>1</td>
</tr>
<tr>
<td>DFACC_RDWR</td>
<td>3</td>
</tr>
</tbody>
</table>

### TABLE 3H
**DFE_* - Error Codes**

<table>
<thead>
<tr>
<th>Definition Name</th>
<th>Definition Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFE_NOERROR</td>
<td>0</td>
</tr>
<tr>
<td>DFE_NONE</td>
<td>0</td>
</tr>
<tr>
<td>DFE_FNF</td>
<td>1</td>
</tr>
<tr>
<td>DFE_DENIED</td>
<td>2</td>
</tr>
<tr>
<td>DFE_ALROPEN</td>
<td>3</td>
</tr>
<tr>
<td>DFE_TOOMANY</td>
<td>4</td>
</tr>
<tr>
<td>DFE_BADNAME</td>
<td>5</td>
</tr>
<tr>
<td>DFE_BADACC</td>
<td>6</td>
</tr>
<tr>
<td>DFE_BADOPEN</td>
<td>7</td>
</tr>
<tr>
<td>DFE_NOTOPEN</td>
<td>8</td>
</tr>
<tr>
<td>DFE_CANTCLOSE</td>
<td>9</td>
</tr>
<tr>
<td>DFE_READERROR</td>
<td>10</td>
</tr>
<tr>
<td>DFE_WRITEERROR</td>
<td>11</td>
</tr>
<tr>
<td>DFE_SEEKERROR</td>
<td>12</td>
</tr>
<tr>
<td>DFE_RDONLY</td>
<td>13</td>
</tr>
<tr>
<td>DFE_BADEEEK</td>
<td>14</td>
</tr>
<tr>
<td>DFE_PUTELEM</td>
<td>15</td>
</tr>
<tr>
<td>DFE_GETELEM</td>
<td>16</td>
</tr>
<tr>
<td>DFE_CANTLINK</td>
<td>17</td>
</tr>
<tr>
<td>DFE_CANTSYNC</td>
<td>18</td>
</tr>
<tr>
<td>Error Code</td>
<td>Number</td>
</tr>
<tr>
<td>----------------</td>
<td>--------</td>
</tr>
<tr>
<td>DFE_BADGROUP</td>
<td>19</td>
</tr>
<tr>
<td>DFE_GROUPSETUP</td>
<td>20</td>
</tr>
<tr>
<td>DFE_PUGROUP</td>
<td>21</td>
</tr>
<tr>
<td>DFE_GROUPWRITE</td>
<td>22</td>
</tr>
<tr>
<td>DFE_DFNULL</td>
<td>23</td>
</tr>
<tr>
<td>DFE_ILLTYPE</td>
<td>24</td>
</tr>
<tr>
<td>DFE_BADLIST</td>
<td>25</td>
</tr>
<tr>
<td>DFE_NOTOFFILE</td>
<td>26</td>
</tr>
<tr>
<td>DFE_SEEDTIC</td>
<td>27</td>
</tr>
<tr>
<td>DFE_MOSDCHTAG</td>
<td>28</td>
</tr>
<tr>
<td>DFE_NOPREDD</td>
<td>29</td>
</tr>
<tr>
<td>DFE_BADTAG</td>
<td>30</td>
</tr>
<tr>
<td>DFE_BADREF</td>
<td>31</td>
</tr>
<tr>
<td>DFE_MOMATCH</td>
<td>32</td>
</tr>
<tr>
<td>DFE_NOTINSET</td>
<td>33</td>
</tr>
<tr>
<td>DFE_BADOFFSET</td>
<td>34</td>
</tr>
<tr>
<td>DFE_CORRUPT</td>
<td>35</td>
</tr>
<tr>
<td>DFE_NONEF</td>
<td>36</td>
</tr>
<tr>
<td>DFE_TOPICD</td>
<td>37</td>
</tr>
<tr>
<td>DFE_CANTMOD</td>
<td>38</td>
</tr>
<tr>
<td>DFE_DFFFIRLES</td>
<td>39</td>
</tr>
<tr>
<td>DFE_BASAID</td>
<td>40</td>
</tr>
<tr>
<td>DFE_OPENAID</td>
<td>41</td>
</tr>
<tr>
<td>DFE_CANTFLUSH</td>
<td>42</td>
</tr>
<tr>
<td>DFE_CANTUPDATE</td>
<td>43</td>
</tr>
<tr>
<td>DFE_CANTHASH</td>
<td>44</td>
</tr>
<tr>
<td>DFE_CANTSELLOD</td>
<td>45</td>
</tr>
<tr>
<td>DFE_CANTSELIRASH</td>
<td>46</td>
</tr>
<tr>
<td>DFE_CANACCESS</td>
<td>47</td>
</tr>
<tr>
<td>DFE_CANTENDACCESS</td>
<td>48</td>
</tr>
<tr>
<td>DFE_TABLEFULL</td>
<td>49</td>
</tr>
<tr>
<td>DFE_NOTINTABLE</td>
<td>50</td>
</tr>
<tr>
<td>DFE_UNSUPPORTED</td>
<td>51</td>
</tr>
<tr>
<td>DFE_HOSPACE</td>
<td>52</td>
</tr>
<tr>
<td>DFE_BADCALL</td>
<td>53</td>
</tr>
<tr>
<td>DFE_BADPTR</td>
<td>54</td>
</tr>
<tr>
<td>DFE_BADLEN</td>
<td>55</td>
</tr>
<tr>
<td>DFE_NOTENOUGH</td>
<td>56</td>
</tr>
<tr>
<td>DFE_NOVALS</td>
<td>57</td>
</tr>
<tr>
<td>DFE_ARGS</td>
<td>58</td>
</tr>
<tr>
<td>DFE_INTERNAL</td>
<td>59</td>
</tr>
<tr>
<td>DFE_HOSENSET</td>
<td>60</td>
</tr>
<tr>
<td>DFE_GENAPP</td>
<td>61</td>
</tr>
<tr>
<td>DFE_UNINIT</td>
<td>62</td>
</tr>
<tr>
<td>DFE_CANTINIT</td>
<td>63</td>
</tr>
<tr>
<td>DFE_CANTSHUTDOWN</td>
<td>64</td>
</tr>
<tr>
<td>DFE_BADDIM</td>
<td>65</td>
</tr>
<tr>
<td>DFE_BADPP</td>
<td>66</td>
</tr>
<tr>
<td>Code</td>
<td>Value</td>
</tr>
<tr>
<td>--------------</td>
<td>-------</td>
</tr>
<tr>
<td>DFE_BADDATATYPE</td>
<td>67</td>
</tr>
<tr>
<td>DFE.BADMCTYPE</td>
<td>68</td>
</tr>
<tr>
<td>DFE.BADMORDER</td>
<td>69</td>
</tr>
<tr>
<td>DFE.BADORDER</td>
<td>70</td>
</tr>
<tr>
<td>DFE_RANGE</td>
<td>71</td>
</tr>
<tr>
<td>DFE.BADCONV</td>
<td>72</td>
</tr>
<tr>
<td>DFE.BADTYPE</td>
<td>73</td>
</tr>
<tr>
<td>DFE_BADSCHEME</td>
<td>74</td>
</tr>
<tr>
<td>DFE_BADMODEL</td>
<td>75</td>
</tr>
<tr>
<td>DFE_BADCODER</td>
<td>76</td>
</tr>
<tr>
<td>DFE.MINIT</td>
<td>77</td>
</tr>
<tr>
<td>DFE_CCODER</td>
<td>78</td>
</tr>
<tr>
<td>DFE_CINIT</td>
<td>79</td>
</tr>
<tr>
<td>DFE_CSDECOD</td>
<td>80</td>
</tr>
<tr>
<td>DFE_CENCODE</td>
<td>81</td>
</tr>
<tr>
<td>DFE_CTERM</td>
<td>82</td>
</tr>
<tr>
<td>DFE_CSEEK</td>
<td>83</td>
</tr>
<tr>
<td>DFE_CINIT</td>
<td>84</td>
</tr>
<tr>
<td>DFE_CANTCOMP</td>
<td>85</td>
</tr>
<tr>
<td>DFE_CANTDECOMP</td>
<td>86</td>
</tr>
<tr>
<td>DFE_MINIT</td>
<td>87</td>
</tr>
<tr>
<td>DFE_MODIN</td>
<td>88</td>
</tr>
<tr>
<td>DFE_BADRIG</td>
<td>89</td>
</tr>
<tr>
<td>DFE_BINTOFIND</td>
<td>90</td>
</tr>
<tr>
<td>DFE_BADATTR</td>
<td>91</td>
</tr>
<tr>
<td>DFE_BADTABLE</td>
<td>92</td>
</tr>
<tr>
<td>DFE_BADDSG</td>
<td>93</td>
</tr>
<tr>
<td>DFE_BADOSG</td>
<td>94</td>
</tr>
<tr>
<td>DFE_VGSIZE</td>
<td>95</td>
</tr>
<tr>
<td>DFE_VTAB</td>
<td>96</td>
</tr>
<tr>
<td>DFE_CANTTOELEM</td>
<td>97</td>
</tr>
<tr>
<td>DFE_BADOWNAME</td>
<td>98</td>
</tr>
<tr>
<td>DFE_BADOCLASS</td>
<td>99</td>
</tr>
<tr>
<td>DFE_BADFIELDS</td>
<td>100</td>
</tr>
<tr>
<td>DFE_NOVES</td>
<td>101</td>
</tr>
<tr>
<td>DFE_SYMSIZE</td>
<td>102</td>
</tr>
<tr>
<td>DFE_BADATTACH</td>
<td>103</td>
</tr>
<tr>
<td>DFE_BADVNAME</td>
<td>104</td>
</tr>
<tr>
<td>DFE_BADVCLASS</td>
<td>105</td>
</tr>
<tr>
<td>DFE_VSNWRITE</td>
<td>106</td>
</tr>
<tr>
<td>DFE_VSNREAD</td>
<td>107</td>
</tr>
<tr>
<td>DFE_BAVSR</td>
<td>108</td>
</tr>
<tr>
<td>DFE_VOCAVCCREASE</td>
<td>109</td>
</tr>
<tr>
<td>DFE_CANTATTACH</td>
<td>110</td>
</tr>
<tr>
<td>DFE_CANTDETACH</td>
<td>111</td>
</tr>
<tr>
<td>DFE_BITREAD</td>
<td>112</td>
</tr>
<tr>
<td>DFE_BITWRITE</td>
<td>113</td>
</tr>
</tbody>
</table>
### Definition Name | Definition Value
--- | ---
DFNT_HDF | 0
DFNT_NATIVE | 4096
DFNT_CUSTOM | 8192
DFNT_LITEND | 16384
DFNT_NONE | 0
DFNT_QUERY | 0
DFNT_VERSION | 1
DFNT_FLOAT32 | 5
DFNT_FLOAT64 | 6
DFNT_DOUBLE | 6
DFNT_FLOAT128 | 7
DFNT_INT8 | 20
DFNT_UINT8 | 21
DFNT_INT16 | 22
DFNT_UINT16 | 23
DFNT_INT32 | 24
DFNT_UINT32 | 25
DFNT_INT64 | 26
DFNT_UINT64 | 27
DFNT_INT128 | 28
DFNT_UINT128 | 29
DFNT_UCHAR8 | 3
DFNT_UCHAR | 3
DFNT_CHAR8 | 4
DFNT_CHAR | 4
DFNT_CHAR16 | 42
DFNT_UCHAR16 | 43
DFNT_NFLOAT32 | 4101
DFNT_NFLOAT64 | 4102
DFNT_NDOUBLE | 4102
DFNT_NFLOAT128 | 4103
DFNT_NINT8 | 4116
DFNT_NUINT8 | 4117
DFNT_NINT16 | 4118
DFNT_NUINT16 | 4119
DFNT_NINT32 | 4120
<table>
<thead>
<tr>
<th>Definition Name</th>
<th>Definition Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFNTF_NONE</td>
<td>0</td>
</tr>
<tr>
<td>DFNTF_HDFDEFAULT</td>
<td>1</td>
</tr>
<tr>
<td>DFNTF_IEEE</td>
<td>1</td>
</tr>
<tr>
<td>DFNTF_VAX</td>
<td>2</td>
</tr>
<tr>
<td>DFNTF_CRAY</td>
<td>3</td>
</tr>
<tr>
<td>DFNTF_PC</td>
<td>4</td>
</tr>
<tr>
<td>DFNTF_CONVEX</td>
<td>5</td>
</tr>
<tr>
<td>DFNTF_VP</td>
<td>6</td>
</tr>
</tbody>
</table>

**TABLE 3J**

DFNTF_* - Floating-point Format Codes
<table>
<thead>
<tr>
<th>Definition Name</th>
<th>Definition Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFTAG_WILDCARD</td>
<td>0</td>
</tr>
<tr>
<td>DFTAG_NULL</td>
<td>1</td>
</tr>
<tr>
<td>DFTAG_LINKED</td>
<td>20</td>
</tr>
<tr>
<td>DFTAG_VERSION</td>
<td>30</td>
</tr>
<tr>
<td>DFTAG_COMPRESSED</td>
<td>40</td>
</tr>
<tr>
<td>DFTAG_VLINKED</td>
<td>50</td>
</tr>
<tr>
<td>DFTAG_VLINKED_DATA</td>
<td>51</td>
</tr>
<tr>
<td>DFTAG_CHUNKED</td>
<td>60</td>
</tr>
<tr>
<td>DFTAG_CHUNK</td>
<td>61</td>
</tr>
<tr>
<td>DFTAG_FID</td>
<td>100</td>
</tr>
<tr>
<td>DFTAG_FD</td>
<td>101</td>
</tr>
<tr>
<td>DFTAG_TID</td>
<td>102</td>
</tr>
<tr>
<td>DFTAG_TD</td>
<td>103</td>
</tr>
<tr>
<td>DFTAG_DIL</td>
<td>104</td>
</tr>
<tr>
<td>DFTAG_STA</td>
<td>105</td>
</tr>
<tr>
<td>DFTAG_NT</td>
<td>106</td>
</tr>
<tr>
<td>DFTAG_MT</td>
<td>107</td>
</tr>
<tr>
<td>DFTAG_ID8</td>
<td>200</td>
</tr>
<tr>
<td>DFTAG_IP8</td>
<td>201</td>
</tr>
<tr>
<td>DFTAG_RI8</td>
<td>202</td>
</tr>
<tr>
<td>DFTAG_CI8</td>
<td>203</td>
</tr>
<tr>
<td>DFTAG_TI8</td>
<td>204</td>
</tr>
<tr>
<td>DFTAG_ID</td>
<td>300</td>
</tr>
<tr>
<td>DFTAG_LUT</td>
<td>301</td>
</tr>
<tr>
<td>DFTAG_RI</td>
<td>302</td>
</tr>
<tr>
<td>DFTAG_CI</td>
<td>303</td>
</tr>
<tr>
<td>DFTAG_RIG</td>
<td>306</td>
</tr>
<tr>
<td>DFTAG_LD</td>
<td>307</td>
</tr>
<tr>
<td>DFTAG_RD</td>
<td>308</td>
</tr>
<tr>
<td>DFTAG_MA</td>
<td>309</td>
</tr>
<tr>
<td>DFTAG_CCN</td>
<td>310</td>
</tr>
<tr>
<td>DFTAG_CFM</td>
<td>311</td>
</tr>
<tr>
<td>DFTAG_AR</td>
<td>312</td>
</tr>
<tr>
<td>DFTAG_DRAM</td>
<td>400</td>
</tr>
<tr>
<td>DFTAG_RUN</td>
<td>401</td>
</tr>
<tr>
<td>DFTAG_XYP</td>
<td>500</td>
</tr>
<tr>
<td>DFTAG_MTO</td>
<td>501</td>
</tr>
<tr>
<td>DFTAG_T14</td>
<td>602</td>
</tr>
<tr>
<td>DFTAG_T105</td>
<td>603</td>
</tr>
<tr>
<td>DFTAG_SDG</td>
<td>700</td>
</tr>
<tr>
<td>DFTAG_SD</td>
<td>701</td>
</tr>
<tr>
<td>DFTAG_SD</td>
<td>702</td>
</tr>
<tr>
<td>DFTAG_SD</td>
<td>703</td>
</tr>
<tr>
<td>DFTAG_SD</td>
<td>704</td>
</tr>
<tr>
<td>DFTAG_SD</td>
<td>705</td>
</tr>
<tr>
<td>Definition Name</td>
<td>Definition Value</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>_HDF_VDATA</td>
<td>-1</td>
</tr>
<tr>
<td>_HDF_VSPACK</td>
<td>0</td>
</tr>
<tr>
<td>_HDF_VSPUNPACK</td>
<td>1</td>
</tr>
<tr>
<td>_HDF_ENTIRE_VDATA</td>
<td>-1</td>
</tr>
<tr>
<td>HDF_APPENDABLE_BLOCK_LEN</td>
<td>4096</td>
</tr>
<tr>
<td>HDF_APPENDABLE_BLOCK_N1M</td>
<td>16</td>
</tr>
</tbody>
</table>

**TABLE 3M**

**MFGR_* - Interlace Mode Codes**

<table>
<thead>
<tr>
<th>Definition Name</th>
<th>Definition Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MFGR_INTERLACE_PIXEL</td>
<td>0</td>
</tr>
<tr>
<td>MFGR_INTERLACE_LINE</td>
<td>1</td>
</tr>
<tr>
<td>MFGR_INTERLACE_COMPONENT</td>
<td>2</td>
</tr>
</tbody>
</table>

**TABLE 3N**

**SD_* - Scientific Data Set Configuration Codes**

<table>
<thead>
<tr>
<th>Definition Name</th>
<th>Definition Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD_UNLIMITED</td>
<td>0</td>
</tr>
<tr>
<td>SD_DIMVAL_BW_COMP</td>
<td>1</td>
</tr>
<tr>
<td>SD_DIMVAL_BW_INCOMP</td>
<td>0</td>
</tr>
<tr>
<td>SD_FILL</td>
<td>0</td>
</tr>
<tr>
<td>SD_HOFILL</td>
<td>256</td>
</tr>
<tr>
<td>SD_RAGGED</td>
<td>-1</td>
</tr>
</tbody>
</table>
### TABLE 3O
**SPECIAL_* - Special Element Identifier Codes**

<table>
<thead>
<tr>
<th>Definition Name</th>
<th>Definition Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL_LINKED</td>
<td>1</td>
</tr>
<tr>
<td>SPECIAL_EXT</td>
<td>2</td>
</tr>
<tr>
<td>SPECIAL_COMP</td>
<td>3</td>
</tr>
<tr>
<td>SPECIAL_VLINKED</td>
<td>4</td>
</tr>
<tr>
<td>SPECIAL_CHUNKED</td>
<td>5</td>
</tr>
</tbody>
</table>

### TABLE 3P
**SUCCEED/FAIL - Routine Return Status Codes**

<table>
<thead>
<tr>
<th>Definition Name</th>
<th>Definition Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCEED</td>
<td>0</td>
</tr>
<tr>
<td>FAIL</td>
<td>-1</td>
</tr>
</tbody>
</table>