HDF User's Guide

Version 4.1r1 • May 1997

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Disclaimer

Authors

Michael Folk - *mfolk@ncsa.uiuc.edu*

Laura Kalman

William Whitehouse - *whitehou@ncsa.uiuc.edu*

NCSA Contacts

Mail user feedback, bug reports, questions, and software and manual suggestions to:

Software Development Division
NCSA HDF
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 the HDF Group's anonymous FTP server:

ftp hdf.ncsa.uiuc.edu

It is also accessible through the HDF Group's World Wide Web home page at

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

Table of Contents

Introduction to HDF 1

1.1 Chapter Overview 1
1.2 What is HDF? 1
1.3 Why Was HDF Created? 3
1.4 High-Level HDF APIs 3
1.4.1 User's Guide Code Examples 4
1.5 NCSA HDF Command-Line Utilities and Visualization Tools 4
1.6 Current HDF Platforms 4

HDF Fundamentals 7

2.1 Chapter Overview 7
2.2 The Hierarchy of HDF Interaction 7
2.3 Data Objects 8
2.3.1 Data Descriptors 8
2.4 File Format 9
2.4.1 Grouping Data Objects in an HDF File 9
2.4.2 Storing Data Objects 10
2.4.2.1 Contiguous Data Elements 10
2.4.2.2 Linked-block Data Elements 10
2.4.2.3 External Data Elements 10
2.5 Header File Information 11
2.5.1 File Access Code Definitions 11
2.5.2 Data Type Definitions 11
2.5.3 Tag Definitions 12
2.5.4 Limit Definitions 13
2.6 Basic Operations on HDF Files 13
2.6.1 File Identifiers 13
2.6.2 Opening HDF Files: Hopen 13
2.6.3 Closing HDF Files: Hclose 14
2.6.4 Determining the Number of Objects with a Specified Tag: Hnumber 14
2.6.5 Getting the HDF Library Version Used to Create a File: Hgetlibversion 14
2.6.6 Locate an Object by its Tag/Reference Number Pair: Hfind 15
2.7 Application Programming Interfaces 15
2.8 Fortran-77 and C Language Issues 15
2.8.1 Fortran-77-to-C Translation 15
2.8.2 Case Sensitivity 16
2.8.3 Name Length 16
2.8.4 Header Files 16
2.8.5 Data Type Specifications 16
2.8.6 Array Specifications 17
2.8.7 Fortran-77, ANSI C and K&R C 17
2.9 Low-Level Interfaces 17
2.9.1 The H Interface 17
2.9.2 The HX Interface 18
2.9.3 The HE Interface 18

Scientific Data Sets (SD API) 19

3.1 Chapter Overview 19

3.2 The SD Scientific Data Set Data Model 19
3.2.1 Required SD SDS Objects 19
3.2.1.1 SDS Array 20
3.2.1.2 SDS Array Name 20
3.2.1.3 Data Type 20
3.2.1.4 Dimensions 20
3.2.2 Optional SD SDS Objects 20
3.2.3 Annotations and the SD Data Model 21
3.3 The SD Scientific Data Set API 21
3.3.1 SD Library Routines 21
3.3.2 SDS Identifiers in the SD Interface 23
3.3.3 Tags in the SD Interface 23
3.4 Programming Model for the SD Interface 23
3.4.1 Establishing Access to Files and Data Sets: SDstart and SDselect 24
3.4.2 Terminating Access to Files and Data Sets: SDendaccess and SDend 25
3.5 Creating and Writing to Simple Scientific Data Sets 26
3.5.1 Creating Scientific Data Sets: SDcreate 26
3.5.2 Writing Data to an SDS Array: SDwritedata 28
3.5.2.1 Filling an Entire Array 31
3.5.2.2 Writing Slabs to an SDS Array 32
3.5.2.3 Appending Data to an SDS Array Using the Unlimited Dimension 37
3.5.3 Compressing SD SDS Array Data: SDsetcompress 40
3.5.3.1 Rules for Writing to a Compressed Data Set 42
3.5.4 External File Operations 42
3.5.4.1 Specifying the Directory Search Path of an External File: HXsetdir 42
3.5.4.2 Specifying the Location of the Next External File to be Created: HXsetcreatedir 43
3.5.4.3 Creating a Data Set in an External File: SDsetexternalfile 43
3.5.4.4 Moving Data to an External File 44
3.6 Reading Data from an SDS Array: SDreaddata 46
3.6.1 Reading Data from an External File 52
3.7 Obtaining Information About SD Data Sets 52
3.7.1 Obtaining Information About the SDSs in a File: SDfileinfo 53
3.7.2 Obtaining Information About a Specific SDS: SDgetinfo 53
3.7.3 Locating a SDS Data Set by Name: SDnametoindex 55
3.7.4 Locating an SDS by Reference Number: SDreftoindex 55
3.7.5 Creating SDS Arrays Containing Variable-Length Data: SDsetnbitdataset 57
3.8 Chunked (or Tiled) Scientific Data Sets 59
3.8.1 Making a Non-Chunked SDS a Chunked SDS: SDsetchunk 59
3.8.2 Setting the Maximum Number of Chunks in the Cache: SDsetchunkcache 60
3.8.3 Writing Data to Chunked SDSs: SDwritechunk and SDwritedata 61
3.8.4 Reading Data From Chunked SDSs: SDreadchunk and SDreaddata 61
3.8.5 Obtaining Information About a Chunked SDS: SDgetchunkinfo 62
3.8.6 Ghost Areas 70
3.9 SD Dimension and Dimension Scale Operations 71
3.9.1 Selecting a Dimension: SDgetdimid 71
3.9.2 Naming a Dimension: SDsetdimname 72
3.9.3 Old and New Dimension Implementations 72
3.9.3.1 Setting the Future Compatibility Mode of a Dimension: SDsetdimval_comp . . . 73
3.9.3.2 Setting the Current Compatibility Mode of a Dimension: SDisdimval_bwcomp 73
3.9.4 Dimension Scales 74
3.9.4.1 Writing Dimension Scales: SDsetdimscale 74
3.9.4.2 Obtaining Dimension Scale and Other Dimension Information: SDdiminfo . . . 74

3.9.4.3 Reading Dimension Scales: SDgetdimscale 75
3.9.5 Distinguishing SDS Arrays from Dimension Scales: SDiscoordvar 78
3.9.6 Dimension Scales for Multiple Data Sets 80
3.10 User-defined Attributes 80
3.10.1 Writing User-defined Attributes: SDsetattr 81
3.10.2 Querying User-defined Attributes: SDfindattr and SDattrinfo 83
3.10.3 Reading User-defined Attributes: SDreadattr 84
3.11 Predefined Attributes 87
3.11.1 Accessing Predefined Attributes 88
3.11.2 SDS String Attributes 89
3.11.2.1 Writing String Attributes: SDsetdatastrs 89
3.11.2.2 Reading String Attributes: SDgetdatastrs 89
3.11.3 Dimension String Attributes 90
3.11.3.1 Writing a Dimension String Attribute: SDsetdimstrs 90
3.11.3.2 Reading a Dimension String Attribute: SDgetdimstrs 90
3.11.4 Range Attributes 91
3.11.4.1 Writing a Range Attribute: SDsetrange 91
3.11.4.2 Reading a Range Attribute: SDgetrange 91
3.11.5 Fill Values 91
3.11.5.1 Writing a Fill Value Attribute: SDsetfillvalue 92
3.11.5.2 Reading a Fill Value Attribute: SDgetfillvalue 92
3.11.5.3 Setting the Fill Mode for all SDSs in the Specified File: SDsetfillmode 92
3.11.6 Calibration Attributes 93
3.11.6.1 Writing Calibrated Data: SDsetcal 93
3.11.6.2 Reading Calibrated Data: SDgetcal 94
3.12 netCDF 95
3.12.1 HDF Interface vs. netCDF Interface 96

Vdatas (VS API) 101

4.1 Chapter Overview 101
4.2 The Vdata Model 101
4.2.1 Records and Fields 102
4.2.2 Conventions 102
4.3 The Vdata API 102
4.3.1 Vdata Library Routines 103
4.3.2 Vdata Identifiers in the Vdata API 104
4.3.3 Programming Model for the Vdata Interface 104
4.3.4 Accessing Files and Vdatas: Hopen, Vstart and VSattach 105
4.3.5 Terminating Access to Vdatas and Files: VSdetach, Vend and Hclose 105
4.4 Creating and Writing to Single-Field Vdatas: VHstoredata and VHstoredatam 107
4.5 Creating and Writing to Multi-Field Vdatas 111
4.5.1 Creating Vdatas 111
4.5.1.1 Creating New Vdatas with VSattach 112
4.5.1.2 Assigning a Vdata Name and Class: VSsetname and VSsetclass 112
4.5.1.3 Defining a Field Within Vdatas: VSfdefine 112
4.5.1.4 Initializing the Fields for Write Access: VSsetfields 113
4.5.1.5 Specifying the Interlace Mode: VSsetinterlace 113
4.5.2 Writing Data to Vdatas 114
4.5.2.1 Resetting the Current Position Within Vdatas: VSseek 114
4.5.2.2 Writing to a Vdata: VSwrite 115

4.5.2.3 Filling Records of Mixed Field Data Types 118
4.6 Reading from Vdatas 122
4.6.1 Selecting the Record to Begin Reading from: VSseek 123
4.6.2 Selecting the Fields to be Read: VSsetfields 123
4.6.3 Reading from the Current Vdata: VSread 123
4.7 Searching for Vdatas in a File 126
4.7.1 Finding All Vdatas that are Not Members of Vgroups: VSlone 126
4.7.2 Searching for the Reference Number of a Vdata: VSgetid 127
4.7.3 Determining a Reference Number from a Vdata Name: VSfind 127
4.7.4 Searching for a Vdata by Field Name: VSfexist 127
4.8 Vdata Attributes 129
4.8.1 Querying the Index of a Vdata Field Given the Field Name: VSfindex 129
4.8.2 Setting the Attribute of a Vdata Field or Vdata: VSsetattr 130
4.8.3 Querying the Total Number of Vdata Attributes: VSnattrs 130
4.8.4 Querying the Number of Attributes of a Vdata or a Vdata Field: VSfnattrs 131
4.8.5 Retrieving the Index of a Vdata Attribute Given the Attribute Name: VSfindattr 131
4.8.6 Querying Information on a Given Vdata Attribute: VSattrinfo 132
4.8.7 Querying the Values of a Given Vdata Attribute: VSgetattr 132
4.8.8 Determining if the Given Vdata is an Attribute: VSisattr 133
4.9 Obtaining Information About a Specific Vdata 135
4.9.1 Obtaining Vdata Information: VSinquire 136
4.9.2 VSQuery Vdata Information Retrieval Routines 138
4.9.3 VS Vdata Information Retrieval Routines 139
4.9.4 VF Field Information Retrieval Routines 140

Vgroups (V API) 141

5.1 Chapter Overview 141
5.2 The Vgroup Data Model 141
5.2.1 Vgroup Names and Classes 141
5.2.2 Vgroup Organization 142
5.2.3 Vgroup Conventions: An Example Using Vsets 142
1.3 The Vgroup API 143
1.3.1 Vgroup Library Routines 143
1.3.2 Vgroup Identifiers in the Vgroup Interface 144
1.4 Programming Model for the Vgroup Interface 145
1.4.1 Accessing Files and Vgroups: Hopen, Vstart and Vattach 145
1.4.2 Terminating Access to Vgroups and Files: Vdetach, Vend and Hclose 146
1.5 Creating and Writing to a Vgroup 146
1.5.1 Creating a Vgroup:Vattach 146
1.5.2 Assigning a Vgroup Name and Class: Vsetname and Vsetclass 147
1.5.3 Inserting Any HDF Data Object Into a Vgroup: Vaddtagref 147
1.5.4 Inserting a Vdata or Vgroup Into a Vgroup: Vinsert 148
1.6 Reading from Vgroups 155
1.6.1 Locating Vgroups in Files 155
1.6.1.1 Locating a Lone Vgroup: Vlone 156
1.6.1.2 Determining the Next Vgroup Identifier: Vgetid 156
1.6.1.3 Determining a Vgroup's Name and Class: Vgetname and Vgetclass 156
1.7 Vgroup Attributes 158
1.7.1 Querying Information on a Given Vgroup Attribute: Vattrinfo 159
1.7.2 Querying the Vgroup Version of a Given Vgroup: Vgetversion 159

1.7.3 Querying the Total Number of Vgroup Attributes: Vnattrs 160
1.7.4 Querying the Values of a Given Vgroup Attribute: Vgetattr 160
1.7.5 Setting the Attribute of a Vgroup: Vsetattr 160
1.7.6 Retrieving the Index of a Vgroup Attribute Given the Attribute Name: Vfindattr 160
1.8 Obtaining Information About the Contents of a Vgroup 163
1.8.1 Returning the Tag/Reference Number Pairs of Vgroup Members: Vntagrefs, Vgettagref and Vgettagrefs 163
1.8.2 Returning Vgroup Member Information: Vinqtagref, Visvg and Visvs 163
1.9 The Vgroup Command-Line Utilities 166
1.9.1 Vshow 166
1.9.2 Vmake 166
1.10 Obsolete Vgroup Interface Routines 166
1.10.1 Determining the Next Vgroup or Vdata Identifier: Vgetnext 166
1.10.2 Determining the Number of Members and Vgroup Name: Vinquire 167
1.11 Vgroup Backward Compatibility Issues 167
1.11.1 Vset Implementation Integrated into the Vgroup Interface 167

8-Bit Raster Images (DFR8 API) 169

6.1 Chapter Overview 169
6.2 The 8-Bit Raster Data Model 169
6.2.1 Required 8-Bit Raster Image Data Set Objects 169
6.2.1.1 8-Bit Raster Image Data Representation 169
6.2.1.2 8-Bit Raster Image Dimension 170
6.2.2 Optional 8-Bit Raster Image Data Set Objects 170
6.2.2.1 Palettes 170
6.2.3 Compression Method 171
6.3 The 8-Bit Raster Image Interface 172
6.3.1 8-Bit Raster Image Library Routines 172
6.4 Writing 8-Bit Raster Images 172
6.4.1 Storing a Raster Image: DFR8putimage and DFR8addimage 173
6.4.2 Adding a Palette to an RIS8 Object: DFR8setpalette 174
6.4.3 Compressing 8-Bit Raster Image Data: DFR8setcompress 176
6.4.4 Specifying the Reference Number of an RIS8: DFR8writeref 179
6.5 Reading 8-Bit Raster Images 180
6.5.1 Reading a Raster Image: DFR8getimage 180
6.5.2 Querying the Dimensions of an 8-Bit Raster Image: DFR8getdims 181
6.5.3 Reading an Image with a Given Reference Number: DFR8readref 182
6.5.4 Specifying the Next 8-Bit Raster Image to be Read: DFR8restart 183
6.6 8-Bit Raster Image Information Retrieval Routines 183
6.6.1 Querying the Total Number of 8-Bit Raster Images: DFR8nimages 183
6.6.2 Determining the Reference Number of the Most-Recently-Accessed 8-Bit Raster Image: DFR8lastref 183
6.6.3 Determining the Reference Number of the Palette of the Most-Recently-Accessed 8-Bit Raster Image: DFR8getpalref 184
6.7 RIS8 Backward Compatability Issues 184
6.7.1 Attribute "long_name" Included in HDF for netCDF Compatibility 184
6.7.2 Raster Image Group Implementation with New RIS8 Tags 184

24-Bit Raster Images (DF24 API) 185

7.1 Chapter Overview 185
7.2 The 24-Bit Raster Data Model 185
7.2.1 Required 24-Bit Raster Image Data Set Objects 185
7.2.1.1 24-Bit Raster Image Data Representation 185
7.2.1.2 24-Bit Raster Image Dimension 186
7.2.2 Optional 24-Bit Raster Image Data Set Objects 186
7.2.2.1 Compression Method 186
7.2.2.2 Interlace Modes 187
7.3 The 24-Bit Raster Interface 188
7.3.1 24-Bit Raster Image Library Routines 188
7.4 Writing 24-Bit Raster Images 188
7.4.1 Writing a 24-Bit Raster Image: DF24putimage and DF24addimage 189
7.4.2 Setting the Interlace Format: DF24setil 190
7.4.3 Compressing Image Data: DF24setcompress and d2sjpeg 192
7.5 Reading 24-Bit Raster Images 194
7.5.1 Reading a Raster Image: DF24getimage 194
7.5.2 Determining the Dimensions of an Image: DF24getdims 194
7.5.3 Modifying the Interlacing of an Image: DF24reqil 195
7.5.4 Reading a 24-Bit Raster Image with a Given Reference Number: DF24readref 196
7.5.5 Specifying that the Next Image Read to be the First 24-Bit Raster Image in the File: DF24restart 197
7.6 24-Bit Raster Image Information Retrieval Routines 197
7.6.1 Querying the Total Number of Images in a File: DF24nimages 197
7.6.2 Querying the Reference Number of the Most Recently Read or Written 24-Bit Raster Image: DF24lastref 198

General Raster Images (GR API) 199

8.1 Chapter Overview 199
8.2 The General Raster Data Model 199
8.2.1 Required General Raster Objects 199
8.2.1.1 Image Array 200
8.2.1.2 Image Array Name 200
8.2.1.3 Pixel Type 200
8.2.1.4 Dimensions 200
8.2.2 Optional General Raster Objects 200
8.3 The General Raster API 201
8.3.1 GR API Routines 201
8.4 Programming Model for the GR Interface 202
8.4.1 Accessing Files and Images: GRstart and GRselect 203
8.4.2 Terminating Access to Files and Images: GRendaccess and GRend 203
8.5 Creating and Writing General Raster Images 204
8.5.1 Creating General Raster Images: GRcreate 204
8.5.2 Setting the Interlace Mode 205
8.5.3 Writing General Raster Images: GRwriteimage 207
8.6 Reading Data from an Image: GRreadimage 210
8.6.1 Reading General Raster Images from an External File 211
8.6.2 Setting the Interlace Mode for a Palette or Image Read 211
8.7 Obtaining Information About Files and General Raster Images 218

8.7.1 Obtaining Information about the Contents of a File: GRfileinfo 218
8.7.2 Obtaining Information About an Image: GRgetiminfo 219
8.7.3 Getting the Index of an Image: GRreftoindex and GRnametoindex 223
8.7.4 Compressing General Raster Images 225
8.7.5 External File Operations Using the GR API 226
8.7.5.1 Creating a General Raster Image in an External File 226
8.7.5.2 Moving General Raster Images to an External File 226
8.7.5.3 Setting the I/O Mode for External General Raster Image Access 227
8.8 General Raster Image Attributes 227
8.8.1 Predefined GR Attributes 227
8.8.2 Setting User-Defined Attributes: GRsetattr 228
8.8.3 Querying User-Defined Attributes: GRfindattr and GRattrinfo 230
8.8.4 Reading User-Defined Attributes: GRgetattr 230
8.9 Reading and Writing Palette Data Using the GR API 233
8.9.1 Obtaining a Palette ID: GRgetlutid 233
8.9.2 Obtaining Palette Information: GRgetlutinfo 233
8.9.3 Retrieving the Reference Number of the Specified Palette: GRluttoref 234
8.9.4 Writing Palette Data: GRwritelut 234
8.9.5 Reading Palette Data: GRreadlut 234
8.10 GR Object Identifier Conversion Functions 238
8.10.1 Mapping an Image Identifier to a Reference Number: GRidtoref 238
8.10.2 Mapping a Reference Number to an Index: GRreftoindex 238

Palettes (DFP API) 239

9.1 Chapter Overview 239
9.2 The Palette Data Model 239
9.3 The Palette API 240
9.3.1 Palette Library Routines 240
9.4 Writing Palettes 240
9.4.1 Writing a Palette: DFPaddpal and DFPputpal 240
9.4.2 Specifying the Reference Number of a Palette: DFPwriteref 242
9.5 Reading a Palette 242
9.5.1 Reading a Palette: DFPgetpal 243
9.5.2 Reading a Palette with a Given Reference Number: DFPreadref 244
9.5.3 Specifying the Next Palette to be Accessed to be the First Palette: DFPrestart 244
9.6 Other Palette Routines 244
9.6.1 Querying the Number of Palettes in a File: DFPnpals 244
9.6.2 Obtaining the Reference Number of the Most Recently Accessed Palette: DFPlastref 245
9.7 Backward Compatibility Issues 245

Annotations (DFAN and AN API) 247

10.1 Chapter Overview 247
10.2 The Annotation Data Model 247
10.2.1 Labels and Descriptions 247
10.2.2 File Annotations 247
10.2.3 Object Annotations 248
10.3 The Single-File Annotation Interface 248

10.3.1 Single-File Annotation Library Routines 249
10.3.2 Tags in the Annotation Interface 249
10.4 Programming Model for the Annotation Interface 250
10.5 Writing Annotations 250
10.5.1 Assigning a File Label: DFANaddfid 251
10.5.2 Assigning a File Description: DFANaddfds 251
10.5.3 Assigning an Object Label: DFANputlabel 252
10.5.4 Assigning an Object Description: DFANputdesc 253
10.6 Reading Annotations 255
10.6.1 Reading a File Label: DFANgetfidlen and DFANgetfid 255
10.6.2 Reading a File Description: DFANgetfdslen and DFANgetfds 256
10.6.3 Reading an Object Label: DFANgetlablen and DFANgetlabel 258
10.6.4 Reading an Object Description: DFANgetdesclen and DFANgetdesc 258
10.7 Maintenance Routines 260
10.7.1 Clearing the DFAN Interface's Internal Structures and Settings: DFANclear 260
10.8 Determining Reference Numbers 261
10.8.1 Determine a Reference Number for the Last Object Accessed: DF*lastref 261
10.8.2 Querying a List of Reference Numbers for a Given Tag: DFANlablist 262
10.8.3 Locate an Object by Its Tag and Reference Number: Hfind 264
10.9 The Multifile Annotation Interface 264
10.9.1 AN Library Routines 264
10.9.2 Type Definitions Used in the Multifile Annotation Interface 265
10.10 Programming Model for the Multifile Annotation Interface 265
10.11 Creating and Writing Annotations Using the AN Interface 266
10.11.1 Creating Annotations 266
10.11.2 Writing an Annotation: ANwriteann 267
10.12 Reading Annotations Using the AN Interface 270
10.12.1 Reading an Annotation: ANreadann 270
10.13 Obtaining Annotation Information Using the AN Interface 271
10.13.1 Selecting an Annotation: ANselect 271
10.13.2 Obtaining General Annotation Information: ANfileinfo 271
10.13.3 Getting the Length of an Annotation: ANannlen 271
10.13.4 Obtaining the Number of Annotations Corresponding to Given Search Criteria: ANnumann 273
10.13.5 Obtaining the List of Annotations Corresponding to Given Search Criteria: ANannlist . 274
10.13.6 Obtaining the Tag/Reference Number Pair From a Specified Annotation Identifiers: ANget_tagref 276
10.13.7 Obtaining the Tag/Reference Number Pair From a Specified Annotation Identifier: ANid2tagref 276
10.13.8 Obtaining the Annotation Identifier From a Specified Tag/Reference Number Pair: ANtagref2id 277
10.13.9 Obtaining an Object Tag From a Specified Annotation Type: ANatype2tag 277
10.13.10 Obtaining an Annotation Type From a Specified Object Tag: ANtag2atype 277

Scientific Data Sets (DFSD API) 279

11.1 Chapter Overview 279
11.2 The DFSD Scientific Data Set Data Model 279
11.2.1 Required DFSD SDS Objects 279
11.2.1.1 Dimensions 280
11.2.2 Optional DFSD SDS Objects 280
11.2.2.1 Dimension Scales 280
11.2.2.2 Predefined Attributes 280
11.3 The Single-File Scientific Data Set Interface 280

11.3.1 DFSD Library Routines 280
11.3.2 File Identifiers in the DFSD Interface 281
11.4 Writing DFSD Scientific Data Sets 281
11.4.1 Creating a DFSD Scientific Data Set: DFSDadddata and DFSDputdata 282
11.4.2 Specifying the Data Type of a DFSD SDS: DFSDsetNT 282
11.4.3 Overwriting Data for a Given Reference Number: DFSDwriteref 284
11.4.4 Writing Several Data Sets: DFSDsetdims and DFSDclear 284
11.4.5 Preventing the Reassignment of DFSD Data Set Attributes: DFSDsetdims 285
11.4.6 Resetting the Default DFSD Interface Settings: DFSDclear 285
11.5 Reading DFSD Scientific Data Sets 286
11.5.1 Reading a DFSD SDS: DFSDgetdata 286
11.5.2 Specifying the Dimensions and Data Type of an SDS: DFSDgetdims and DFSDgetNT . . 287
11.5.3 Determining the Number of DFSD Data Sets: DFSDndatasets and DFSDrestart 288
11.5.4 Obtaining Reference Numbers of DFSD Data Sets: DFSDreadref and DFSDlastref 289
11.6 Slabs in the DFSD Interface 290
11.6.1 Accessing Slabs: DFSDstartslab and DFSDendslab 290
11.6.2 Writing Slabs: DFSDwriteslab 290
11.6.3 Reading Slabs: DFSDreadslab 291
11.7 Predefined Attributes and the DFSD Interface 292
11.7.1 Writing Data Set Attributes 293
11.7.1.1 Assigning String Attributes to an SDS: DFSDsetlengths and DFSDsetdatastrs . 293
11.7.1.2 Assigning Value Attributes to a DFSD SDS: DFSDsetfillvalue, DFSDsetrange, DFSDset-cal 294
11.7.2 Reading DFSD Data Set Attributes 297
11.7.2.1 Reading Data Set Attributes: DFSDgetdatalen and DFSDgetdatastrs 297
11.7.2.2 Reading the Value Attributes of a DFSD Data Set: DFSDgetfillvalue and DFSDgetcal 298
11.7.3 Writing the Dimension Attributes of a DFSD SDS 299
11.7.3.1 Writing the String Attributes of a Dimension: DFSDsetlengths and DFSDsetdimstrs 299
11.7.3.2 Writing a Dimension Scale of a DFSD SDS: DFSDsetdimscale 300
11.7.4 Reading the Dimension Attributes of a DFSD SDS 301

Error Reporting 303

12.1 Chapter Overview 303
12.2 The HDF Error Reporting API 303
12.3 Error Reporting in HDF 303
12.3.1 Writing Errors to a File: HEprint 304
12.3.2 Returning the Code of the Nth Most Recent Error: HEvalue 304
12.3.3 Returning the Description of an Error Code: HEstring 305

HDF Performance Issues 311

13.1 Chapter Overview and Introduction 311
13.2 Examples of HDF Performance Enhancement 311
13.2.1 One Large SDS Versus Several Smaller SDSs 312
13.2.2 Sharing Dimensions Between Scientific Data Sets 313
13.2.3 Setting the Fill Mode 315
13.2.4 Disabling "Fake" Dimension Scale Values in Large One-Dimensional Scientific Data Sets 316
13.3 Data Chunking 318

13.3.1 What is Data Chunking? 318
13.3.2 Writing Concerns and Reading Concerns in Chunking 319
13.3.3 Chunking without Compression 319
13.3.4 Chunking with Compression 322
13.3.5 Effect of Chunk Size on Performance 324
13.3.6 How Insufficient Chunk Cache Space can Impair Chunking Performance 324

HDF Command-Line Utilities 331

14.1 Chapter Overview 331
14.2 The HDF Command-Line Utilities 331
14.2.1 Listing Basic Information About an HDF File: hdfls 332
14.2.1.1 General Description 332
14.2.1.2 Command-Line Syntax 332
14.2.1.3 Examples 332
14.2.2 Editing the Contents of an HDF File: hdfed 333
14.2.2.1 General Description 333
14.2.2.2 Command-Line Syntax 334
14.2.3 Converting Floating-Point Data to SDS or RIS8: fp2hdf 342
14.2.3.1 General Description 342
14.2.3.2 Command-Line Syntax 343
14.2.3.3 Examples 344
14.2.4 Converting Several RIS8 Images to One 3D SDS: ristosds 344
14.2.4.1 General Description 344
14.2.4.2 Command-Line Syntax 344
14.2.4.3 Examples 345
14.2.5 Converting 8-Bit Raster Images to the HDF Format: r8tohdf 345
14.2.5.1 General Description 345
14.2.5.2 Command-Line Syntax 345
14.2.5.3 Examples 345
14.2.6 Extracting 8-Bit Raster Images and Palettes from HDF Files: hdftor8 346
14.2.6.1 General Description 346
14.2.6.2 Command-Line Syntax 346
14.2.6.3 Examples 347
14.2.7 Compressing RIS8 Images in an HDF File: hdfcomp 347
14.2.7.1 General Description 347
14.2.7.2 Command-Line Syntax 347
14.2.7.3 Examples 347
14.2.8 Converting 24-Bit Raw Raster Images to RIS8 Images: r24hdf8 347
14.2.8.1 General Description 347
14.2.8.2 Command-Line Syntax 347
14.2.8.3 Examples 348
14.2.9 Converting an HDF RIS24 Image to an HDF RIS8 Image: hdf24hdf8 348
14.2.9.1 General Description 348
14.2.9.2 Command-Line Syntax 348
14.2.10 Converting Raw Palette Data to the HDF Palette Format: paltohdf 348
14.2.10.1 General Description 348
14.2.10.2 Command-Line Syntax 348
14.2.11 Extracting Palette Data from an HDF File: hdftopal 349
14.2.11.1 General Description 349
14.2.11.2 Command-Line Syntax 349

14.2.12 Compressing an HDF File: hdfpack 349
14.2.12.1 General Description 349
14.2.12.2 Command-Line Syntax 349
14.2.12.3 Examples 349
14.2.13 Displaying Vdata Information: vshow 350
14.2.13.1 General Description 350
14.2.13.2 Command-Line Syntax 350
14.2.13.3 Examples 350
14.2.14 Displaying General Information About the Contents of an HDF File: hdp 350
14.2.14.1 General Description 350
14.2.14.2 Command-Line Syntax 350
14.2.15 The HDF User-Contributed Utilities 352

Appendices 355

Appendix A: NCSA HDF Tags 355
Appendix B: HDF Installation Overview 360

List of Tables

TABLE 2A File Access Code Definitions 11
TABLE 2B Data Type Definitions 11
TABLE 2C Tag Definitions 12
TABLE 2D Limit Definitions 13
TABLE 2E Hopen Parameter List 14
TABLE 2F Hclose Parameter List 14
TABLE 2G Hnumber Parameter List 14
TABLE 2H Hgetlibversion Parameter List 15
TABLE 2I Hfind Parameter List 15
TABLE 2J Data Type Definitions 16
TABLE 2K Some H Interface Routines 17
TABLE 3A SD Library Routines 22
TABLE 3B SDstart, SDselect, SDend and SDendaccess Parameter List 25
TABLE 3C SDcreate Parameter List 27
TABLE 3D SDwritedata Parameter List 30
TABLE 3E SDsetcompress Parameter List 41
TABLE 3F HXsetdir Parameter List 43
TABLE 3G HXsetcreatedir Parameter List 43
TABLE 3H SDsetexternalfile Parameter List 44
TABLE 3I SDreaddata Parameter List 47
TABLE 3J SDfileinfo and SDgetinfo Parameter List 53
TABLE 3K SDnametoindex and SDreftoindex Parameter List 56
TABLE 3L SDsetnbitdataset Parameter List 58
TABLE 3M SDsetchunk Parameter List 59
TABLE 3N SDsetchunkcache Parameter List 60
TABLE 3O SDwritechunk Parameter List 61
TABLE 3P SDreadchunk Parameter List 62
TABLE 3Q SDgetchunkinfo Parameter List 62
TABLE 3R SDsetdimname Parameter List 72
TABLE 3S SDsetdimval_comp Parameter List 73
TABLE 3T SDisdimval_bwcomp Parameter List 74
TABLE 3U SDgetdimid, SDsetdimname, SDsetdimscale, SDdiminfo and SDgetdimscale Parameter List 75
TABLE 3V SDiscoordvar Parameter List 78
TABLE 3W SDsetattr, SDfindattr, SDattrinfo and SDreadattr Parameter List 84
TABLE 3X Predefined Attribute List 87
TABLE 3Y Predefined Attribute Parameter List 88
TABLE 3Z SDsetdatastrs and SDgetdatastrs Parameter List 89
TABLE 3AA SDsetdimstrs and SDgetdimstrs Parameter List 90
TABLE 3AB SDsetrange and SDgetrange Parameter List 91
TABLE 3AC SDsetfillvalue, SDgetfillvalue and SDsetfillmode Parameter List, ... 93
TABLE 3AD SDsetcal and SDgetcal Parameter List 94
TABLE 3AE Summary of HDF and XDR File Compatibility for the HDF and netCDF APIs 96
TABLE 3AF NC Interface Routine Calls and Their SD Equivalents 97
TABLE 4A Vdata Interface Routines 103
TABLE 4B Vstart, VSattach, VSdetach and Vend Parameter List 106
TABLE 4C VHstoredataand VHstoredatam Parameter List 109
TABLE 4D Predefined Data Types and Field Names for Vdata Fields 112
TABLE 4E VSsetname, VSsetclass, VSfdefine, VSsetfields and VSsetinterlace Parameter List . . 113
TABLE 4F VSseek and VSwrite Parameter List 116
TABLE 4G VSseek, VSsetfields and VSread Parameter List 124
TABLE 4H VSlone, VSgetid, VSfind and VSfexist Parameter List 127
TABLE 4I VSfindex Parameter List 130
TABLE 4J VSsetattr Parameter List 130

TABLE 4K VSnattrs Parameter List 130
TABLE 4L VSfnattrs Parameter List 131
TABLE 4M VSfindattr Parameter List. 131
TABLE 4N VSattrinfo Parameter List. 132
TABLE 4O VSgetattr Parameter List 132
TABLE 4P VSisattr Parameter List. 133
TABLE 4Q VSinquire Parameter List 136
TABLE 4R VSQuery Routine Parameter List. 138
TABLE 4S VSelts, VSgetclass, VSgetfields, VSgetinterlace, VSgetname and VSsizeof Parameter List139
TABLE 4T VFfieldname, VFfieldesize, VFfieldisize, VFnfields, VFfieldorder and VFfieldtype Parameter List140
TABLE 5A Vgroup Interface Routines 144
TABLE 5B Vstart, Vattach, Vdetach and Vend Parameter List. 146
TABLE 5C Vsetname, Vsetclass, Vaddtagref and Vinsert Parameter List 148
TABLE 5D Vlone, Vgetid, Vgetname and Vgetclass Parameter List 156
TABLE 5E Vattrinfo Parameter List 159
TABLE 5F Vgetversion Parameter List 159
TABLE 5G Vnattrs Parameter List 160
TABLE 5H Vgetattr Parameter List. 160
TABLE 5I Vsetattr Parameter List. 160
TABLE 5J Vfindattr Parameter List. 161
TABLE 5K Vgetname, Vgetclass, Visvg and Visvs Parameter List 164
TABLE 5L Vgetnext and Vinquire Parameter List 167
TABLE 6A 8-Bit Raster Image Compression Method List. 171
TABLE 6B DFR8 Library Routines 172
TABLE 6C DFR8putimage and DFR8addimage Parameter List 173
TABLE 6D DFR8setpalette Parameter List. 175
TABLE 6E DFR8setcompress Parameter List 176
TABLE 6F DFR8writeref Parameter List 180
TABLE 6G DFR8getdims and DFR8getimage Parameter List. 181
TABLE 6H DFR8readref Parameter List. 183
TABLE 6I DFR8nimages Parameter List. 183
TABLE 6J DFR8nimages Parameter List. 184
TABLE 7A 24-Bit Raster Image Compression Method List. 187
TABLE 7B 24-Bit Raster Image Interlace Format 187
TABLE 7C DF24 Library Routines. 188
TABLE 7D DF24putimage and DF24addimage Parameter List 189
TABLE 7E DF24setil and DF24setcompress Parameter List 192
TABLE 7F DF24getimage, DF24getdims and DF24reqil Parameter List .195
TABLE 7G DF24readref Parameter List 197
TABLE 7H DF24restart Parameter List. 197
TABLE 7I DF24nimages Parameter List 198
TABLE 7J DF24lastref Parameter List. 198
TABLE 8A GR Library Routines 201
TABLE 8B GRstart, GRselect, GRend and GRendaccess Parameter List .204
TABLE 8C GRcreate Parameter List. 205
TABLE 8D GRwriteimage Parameter List 208
TABLE 8E GRreadimage, GRreqlutil and GRreqimageil Parameter List .211
TABLE 8F GRfileinfo and GRgetiminfo Parameter List 219
TABLE 8G GRreftoindex and GRnametoindex Parameter List 223
TABLE 8H GRsetcompress Parameter List. 225
TABLE 8I GRsetexternalfile Parameter List 226

TABLE 8J GRsetaccesstype Parameter List . . . 227
TABLE 8K GRsetattr, GRfindattr, GRattrinfo and GRgetattr Parameter List . . . 231
TABLE 8L GRluttoref Parameter List . . . 234
TABLE 8M GRgetlutid, GRgetlutinfo, GRwritelut and GRreadlut Parameter List . . . 235
TABLE 8N GRidtoref and GRreftoindex Parameter List. . . . 238
TABLE 9A DFP Library Routines . . . 240
TABLE 9B DFPputpal and DFPaddpal Parameter List . . . 241
TABLE 9C DFPwriteref Parameter List . . . 242
TABLE 9D DFPgetpal Parameter List . . . 243
TABLE 9E DFPreadref Parameter List . . . 244
TABLE 9F DFPnpals Parameter List. . . . 245
TABLE 10A DFAN Library Routines . . . 249
TABLE 10B List of Annotation Interface Tags in HDF Versions 2.0, 3.0 and 4.0 . . . 249
TABLE 10C DFANaddfid and DFANaddfds Parameter List. . . . 251
TABLE 10D DFANputlabel and DFANputdesc Parameter List . . . 253
TABLE 10E DFANgetfidlen, DFANgetfid, DFANgetfdslen and DFANgetfds Parameter List. . . . 256
TABLE 10F DFANgetlablen, DFANgetlabel, DFANgetdesc and DFANgetdesclen Parameter List 259
TABLE 10G DFANclear Parameter List . . . 260
TABLE 10H List and Descriptions of the DF*writeref and DF*lastref Routines . . . 261
TABLE 10I DFANlablist Parameter List . . . 263
TABLE 10J AN Library Routines . . . 265
TABLE 10K ANstart, ANcreate, ANcreatef, ANendaccess and ANend Parameter List. . . . 267
TABLE 10L ANwriteann Parameter List. . . . 268
TABLE 10M ANreadann Parameter List . . . 270
TABLE 10N ANselect Parameter List . . . 271
TABLE 10O ANfileinfo Parameter List . . . 271
TABLE 10P ANannlen Parameter List . . . 272
TABLE 10Q ANnumann Parameter List . . . 274
TABLE 10R ANannlist Parameter List . . . 274
TABLE 10S ANget_tagref Parameter List . . . 276
TABLE 10T ANid2tagref Parameter List . . . 276
TABLE 10U ANtagref2id Parameter List . . . 277
TABLE 10V ANatype2tag Parameter List . . . 277
TABLE 10W ANtag2atype Parameter List . . . 277
TABLE 11A DFSD Library Routines . . . 281
TABLE 11B DFSDadddata and DFSDputdata Parameter List. . . . 282
TABLE 11C DFSDsetNT and DFSDwriteref Parameter List . . . 284
TABLE 11D DFSDsetdims Parameter List . . . 285
TABLE 11E DFSDclear Parameter List . . . 286
TABLE 11F DFSDgetdata Parameter List. . . . 286
TABLE 11G DFSDgetNT and DFSDgetdims Parameter List . . . 287
TABLE 11H DFSDreadref Parameter List. . . . 289
TABLE 11I DFSDstartslab Parameter List. . . . 290
TABLE 11J DFSDwriteslab Parameter List . . . 291
TABLE 11K DFSDreadslab Parameter List. . . . 292
TABLE 11L DFSDsetlengths and DFSDsetdatastrs Parameter List . . . 293
TABLE 11M DFSDsetfillvalue, DFSDsetrange and DFSDsetcal Parameter List . . . 294
TABLE 11N DFSDgetdatalen and DFSDgetdatastrs Parameter List. . . . 297
TABLE 11O DFSDgetfillvalue, DFSDgetcal and DFSDgetrange Parameter List. . . . 299
TABLE 11P DFSDsetlengths and DFSDsetdimstrs Parameter List . . . 300
TABLE 11Q DFSDsetdimscale Parameter List . . . 301
TABLE 11R DFSDgetdimlen, DFSDgetdimstrs and DFSDgetdimscale Parameter List . . . 301

TABLE 12A Error Reporting Routine List 303
TABLE 12B HDF Error Codes 305
TABLE 13A Results of the Write Operation to 1,000 1 x 1 x 1 Element Scientific Data Sets 312
TABLE 13B Results of the Write Operation to One 10 x 10 x 10 Element Scientific Data Set 313
TABLE 13C Results of the Write Operation to 1,000 1 x 1 x 1 Element Scientific Data Sets 314
TABLE 13D Results of the Write Operation to 1,000 1 x 1 x 1 SDSs with Shared Dimensions 315
TABLE 13E Results of the Write Operation to the 50 10 x 10 x 10 SDSs with the Fill Value Write Enabled 316
TABLE 13F Results of the Write Operation to the 50 SDSs with the Fill Value Write Disabled . . .316
TABLE 13G Results of the SDS Write Operation with the New and Old Dimension Scales. 317
TABLE 13H Results of the SDS Write Operation With Only the New Dimension Scale 318
TABLE 14A The HDF Command-Line Utilities. 331
TABLE 14B The hdfed Command Set 335
TABLE 14C The hdp Command Set 350
TABLE 14D HDF User-Contributed Utilities 353
TABLE A The HDF Utility Tags 356
TABLE B The HDF General Raster Image Tags. 357
TABLE C The HDF Composite Image Tags 357
TABLE D The HDF Scientific Data Set Tags 358
TABLE E The HDF Vset Tags 358
TABLE F The Obsolete HDF Tags 358

List of Examples

Introduction to HDF

HDF Fundamentals

Scientific Data Sets (SD API)

EXAMPLE 1 Accessing and Closing an SDS 25
EXAMPLE 2 Creating an Empty SDS 27
EXAMPLE 3 Creating and Writing to an SDS 31
EXAMPLE 4 Writing a Slab of Data to an SDS 33
EXAMPLE 5 Altering Values Within an Array Data Set 35
EXAMPLE 6 Appending Data to an SDS Array Using an Unlimited Dimension 37
EXAMPLE 7 Writing SDS Data to an HDF File Starting at an Offset 45
EXAMPLE 8 Reading an Entire SDS 47
EXAMPLE 9 Reading a Subset of an SDS Array 48
EXAMPLE 10 Sampling SDS Data 50
EXAMPLE 11 Printing Data Set Names 53
EXAMPLE 12 Searching for the Index of an SDS 56
EXAMPLE 13 Writing and Reading Chunked Data Using SDwritechunk and SDreaddata 62
EXAMPLE 14 Writing and Reading Chunked Data Using SDwritedata and SDreaddata 65
EXAMPLE 15 Writing and Reading Chunked Data Using SDwritedata and SDreadchunk 66
EXAMPLE 16 Writing and Reading Compressed Chunked Data Using SDwritechunk and SDreaddata 68
EXAMPLE 17 Writing Dimension Information 76
EXAMPLE 18 Retrieving SDS Information from an HDF File 78
EXAMPLE 19 Setting Attribute Values 82
EXAMPLE 20 Retrieving Attribute Information 85
EXAMPLE 21 Calibrating Data 94

Vdatas (VS API)

EXAMPLE 1 Accessing a Vdata in an HDF File 106
EXAMPLE 2 Creating Single-Field Vsets 109
EXAMPLE 3 Writing Data to a New Vdata 116
EXAMPLE 4 Writing Packed Vdata Data to an HDF File 119
EXAMPLE 5 Reading Data from a Vdata 124
EXAMPLE 6 Locating a Vdata from a Specified Field Name 127
EXAMPLE 7 Attaching and Querying Vdata Attributes 133
EXAMPLE 8 Obtaining Vdata Information 136

Vgroups (V API)

EXAMPLE 1 Creating a Vgroup Containing a Raster Image 148
EXAMPLE 2 Inserting Three Vdatas Into a Vgroup to Create a Vset 150
EXAMPLE 3 Adding a Scientific Data Set to a Vgroup 154
EXAMPLE 4 Printing Vgroup Reference Numbers 156
EXAMPLE 5 Attaching and Querying Vgroup Attributes 161
EXAMPLE 6 Extracting the Tag/Reference Number Pairs of Vgroup Members 164

8-Bit Raster Images (DFR8 API)

EXAMPLE 1 Writing an 8-Bit Raster Image to an HDF File 173
EXAMPLE 2 Writing a Palette and an Image in RIS8 Format 175

EXAMPLE 3 Writing a Set of Compressed 8-Bit Raster Images 177
EXAMPLE 4 Compressing and Writing a 8-Bit Raster Image 178
EXAMPLE 5 Reading an 8-Bit Raster Image 181

24-Bit Raster Images (DF24 API)

EXAMPLE 1 Writing a 24-Bit Raster Image to an HDF File 189
EXAMPLE 2 Writing 24-Bit Raster Images Using Scan-plane Interlacing 191
EXAMPLE 3 Compressing and Writing a 24-Bit Raster Image 193
EXAMPLE 4 Reading a 24-Bit Raster Image from an HDF File 195

General Raster Images (GR API)

EXAMPLE 1 Creating and Accessing a General Raster Data Set 205
EXAMPLE 2 Creating and Writing an Image 208
EXAMPLE 3 Reading an Image 212
EXAMPLE 4 Reading a Subset of an Image Array 214
EXAMPLE 5 Sampling Image Data 216
EXAMPLE 6 Retrieving Image Information 220
EXAMPLE 7 Printing Image Names 221
EXAMPLE 8 Searching for an Image Index 223
EXAMPLE 9 Setting File and Image Attribute Values 228
EXAMPLE 10 Retrieving Image Attribute Information 231
EXAMPLE 11 Reading and Writing a Palette and Obtaining Palette Information 235

Palettes (DFP API)

EXAMPLE 1 Writing a Palette 241
EXAMPLE 2 Reading a Palette 243

Annotations (DFAN and AN API)

EXAMPLE 3 Writing a File Label and a File Description 251
EXAMPLE 4 Writing an Object Label and Description to a Scientific Data Set 253
EXAMPLE 5 Reading a File Label and a File Description 257
EXAMPLE 6 Reading an Object Label and Description 259
EXAMPLE 7 Getting a List of Labels for All Scientific Data Sets 263
EXAMPLE 8 Adding a File Label Using the AN Interface 268
EXAMPLE 9 Reading File Labels 272
EXAMPLE 10 Returning the Number of and the List of Object Descriptions 274

Scientific Data Sets (DFSD API)

EXAMPLE 1 Creating and Writing to a DFSD Scientific Data Set 283
EXAMPLE 2 Reading from a DFSD Scientific Data Set 287
EXAMPLE 3 Assigning Predefined String Attributes to a File 295
EXAMPLE 4 Reading a Data Set and its Attribute Record 297

Error Reporting

EXAMPLE 1 Writing Errors to a Console Window 305

HDF Performance Issues

HDF Command-Line Utilities

Appendices

Chapter 1

Introduction to HDF

1.1 Chapter Overview

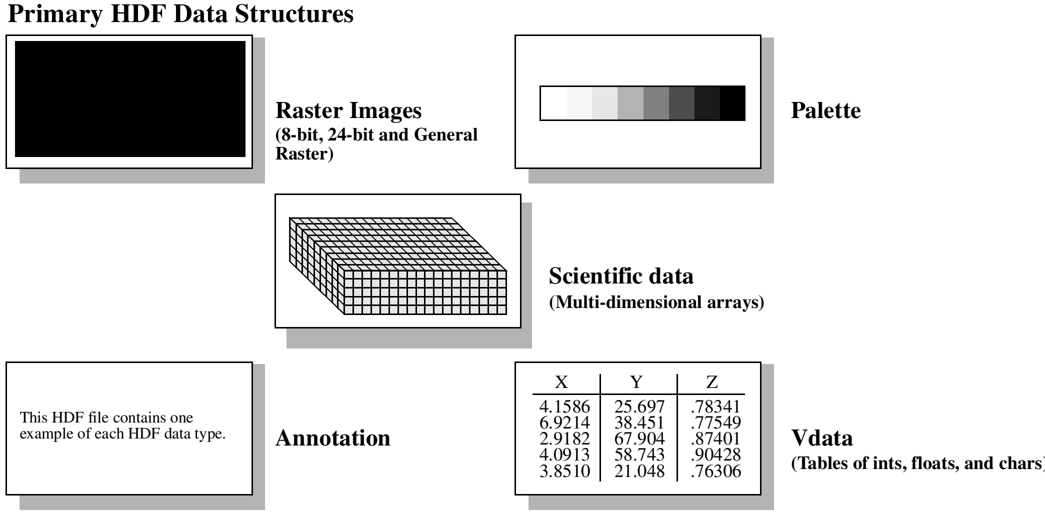
This chapter provides a general description of HDF including its indigenous object structures, application programming interface and accompanying command-line utilities. It also provides a short discussion of HDF's original purpose and philosophy and concludes with a list of the platforms HDF has been ported to.

1.2 What is HDF?

The ***Hierarchical Data Format***, or ***HDF***, is a multi-object file format for sharing scientific data in a distributed environment. HDF was created at the National Center for Supercomputing Applications to serve the needs of diverse groups of scientists working on projects in many fields. HDF was designed to address many requirements for storing scientific data, including:

- Support for the types of data and metadata commonly used by scientists.
- Efficient storage of and access to large data sets.
- Platform independence.
- Extensibility for future enhancements and compatibility with other standard formats.

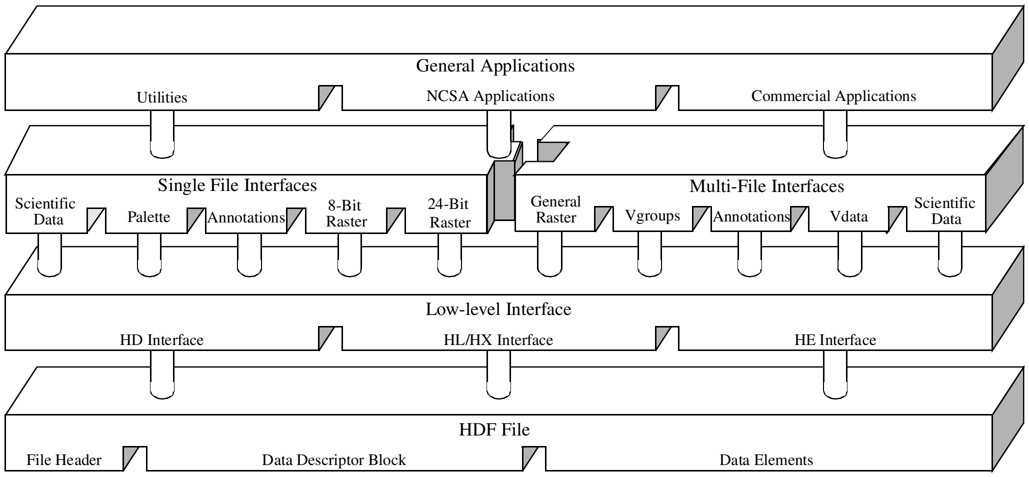
FIGURE 1a **Primary HDF Data Structures**



Five of the primary data objects supported by HDF are illustrated in Figure 1a. A sixth, the vgroup object, does not contain data and is designed for the purpose of grouping the other five primary data objects within an HDF file.

HDF is more than a file format. It also consists of supporting software that make it easy to store, retrieve, visualize, analyze, and manage data in HDF files. HDF can be viewed as several interactive levels as illustrated in the following figure.

FIGURE 1b **The Three Levels of Interaction with the HDF File Format**



At its lowest level, HDF is a physical file format for storing scientific data. At its highest level, HDF is a collection of utilities and applications for manipulating, viewing, and analyzing data in HDF files. Between these levels, HDF is a software library that provides high-level APIs and a low-level data interface.

The basic interface layer, or the ***low-level interface***, is reserved for software developers. It was designed for direct file I/O of data streams, error handling, memory management, and physical storage. It is essentially a software toolkit for skilled programmers who wish to make HDF do something more than what is currently available through the higher-level interfaces. Low-level routines are only available in C.

The HDF ***application programming interfaces,*** or ***APIs,*** include several independent sets of routines, with each set specifically designed to simplify the process of storing and accessing one type of data. These APIs are represented in Figure 1b as the second layer from the top. Although each interface requires programming, all the low-level details can be ignored. In most cases, all one must do is make the correct function call at the correct time, and the interface will take care of the rest. Most HDF API routines are available in both Fortran-77 and C.

The routines that make up the low-level interface and the APIs are available in the NCSA HDF libraries. Source code for the HDF libraries, as well as binaries for some platforms, is in the public domain and is on the NCSA ftp server at `hdf.ncsa.uiuc.edu`.

On the highest "general applications" level, HDF includes ***command-line utilities*** for managing and viewing HDF files, ***NCSA applications*** that support data visualization and analysis, and a variety of ***third-party applications***. HDF utilities are included in the NCSA HDF distribution. Applications supported by NCSA, as well as applications contributed by members of the world-wide HDF user community are freely available on the NCSA ftp server.

1.3 Why Was HDF Created?

Scientists commonly generate and process data files on several different machines, use various software packages to process files and share data files with others who use different machines and software. Also, they may include different kinds of information within one particular file, or within a group of files, and the mixture of these different kinds of information may vary from one file to another. Files may be conceptually related but physically separated. For example, some data may be dispersed among different files and some in program code. It is also possible that data may be related only in the scientist's conception of the data; no physical relationship may exist.

HDF addresses these problems by providing a general-purpose file structure that does the following:

- Provides the mechanism for programs to obtain information about the data in a file from the file itself, rather than from another source.
- Lets you store mixtures of different data types and related information in more than one file, even when the files are processed by the same application program.
- Standardizes the formats and descriptions of many types of commonly-used datasets, such as raster images and multidimensional arrays.
- Encourages the use of a common data format by all machines and programs that produce files containing a specific dataset.
- Can be adapted to accommodate virtually any kind of data by defining new tags or new combinations of tags.

HDF files are ***self-describing***. For each data object in an HDF file, there is information about the type of data, the amount of data, its dimensions, and its location in the file. "Self-description" means that many types of data can be included within an HDF file. For example, it is possible to have symbolic, numerical and graphical data within one HDF file.

1.4 High-Level HDF APIs

The HDF I/O library consists of C and Fortran-77 routines for accessing objects and associated information. Although there is some overlap among object types, in most cases an API operates on data of only one type. Therefore, you need only familiarize yourself with the APIs specific to your needs in order to access data in an HDF file. The names of the HDF APIs designed to access the seven object types are:

- **DFR8 API** Stores, manages and retrieves 8-bit raster images, their dimensions and palettes in one file. It is described in Chapter 6, titled *8-bit Raster Images (DFR8 API)*.
- **DFP API** Stores and retrieves 8-bit palettes in one file. It is described in Chapter 9, titled *Palettes (DFP API)*.
- **DF24 API** Stores, manages and retrieves 24-bit images and their dimensions in one file. It is described in Chapter 7, titled *24-bit Raster Images (DF24 API)*.
- **SD API** Stores, manages and retrieves multi-dimensional arrays of integer or floating-point data, along with their dimensions and attributes, in more than one file. It is described in Chapter 3, titled *Scientific Data Sets (SD API)*.
- **DFSD API** Stores, manages and retrieves multi-dimensional arrays of integer or floating-point data, along with their dimensions and

attributes, in one file. It is described in Chapter 11, titled *Scientific Data Sets (DFSD API)*.

- **AN API** — Stores, manages and retrieves text strings used to describe a file or any of the data elements it contains. It is described in Chapter 10, titled *Annotations (AN API)*.
- **VS API** — Stores, manages and retrieves multivariate data stored as records in a table. It is described in Chapter 4, titled *Vdatas (VS API)*.
- **V API** — Creates groups of any primary HDF object type. It is described in Chapter 5, titled *Vgroups (V API)*.
- **GR API** — Stores, manages and retrieves raster images of several bit lengths, their dimensions and palettes in more than one file. It can also manipulate unattached palettes in more than one file. It is described in Chapter 8, titled *General Raster Images (GR API)*.

As these interfaces are the tools used to read and write HDF files they are the primary focus of this manual.

1.4.1 User's Guide Code Examples

Copies of the C and Fortran-77 User's Guide examples, in ASCII format, are located on the HDF ftp server at `hdf.ncsa.uiuc.edu` in the "/pub/HDF/examples/Users_Guide" directory.

1.5 NCSA HDF Command-Line Utilities and Visualization Tools

HDF application software fall within the following three categories:

1. The Fortran-77 and C APIs described above.
2. Scientific visualization and analysis tools that read and write HDF files.
3. Command-line utilities that operate directly on HDF files.

Scientific visualization and analysis software that can read and write HDF files is available. This software includes NCSA tools such as Collage, user-developed software, and commercial packages. The use of HDF files guarantees the interoperability of the such tools. Some tools operate on raster images, others on color palettes. Some use images, others color palettes, still others data and annotations, and so forth. HDF provides the range of data types that these tools need, in a format that allows different tools with different data requirements to operate on the same files without confusion.

The HDF ***command-line utilities*** are application programs that can be executed by entering them at the command prompt, like UNIX commands. They make it possible for you to perform common operations on HDF files for which you would normally have to write your own program. The HDF utilities are described in detail in Chapter 13, titled *HDF Command-Line Utilities*.

1.6 Current HDF Platforms

The HDF library and utilities are maintained on a number of different machines and operating systems. Refer to the 'INSTALL' file in the top-level directory of the HDF distribution for this list.

Chapter 2

HDF Fundamentals

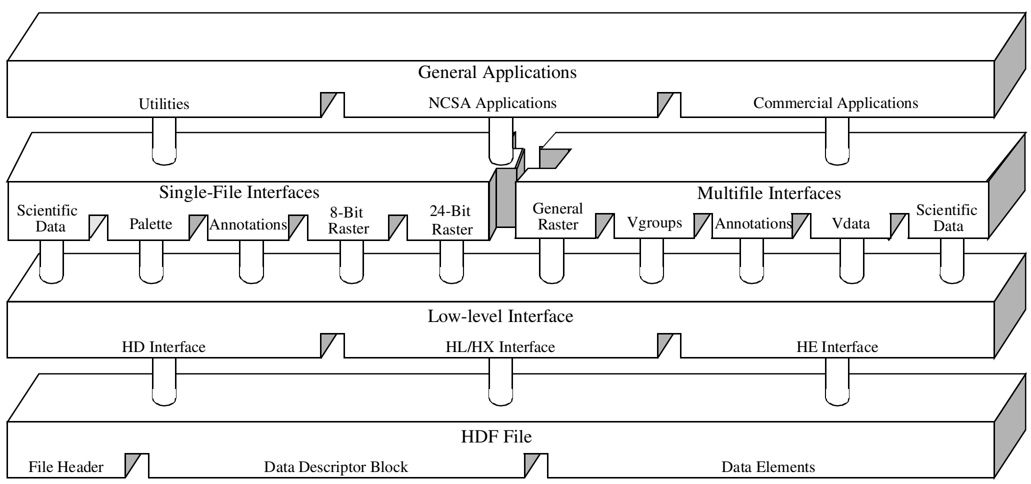
2.1 Chapter Overview

In this chapter, our description of HDF in Chapter 1, titled *Introduction to HDF*, is expanded to include a description of the hierarchical structure of HDF interaction with HDF file objects, the physical format of HDF files, the low-level HDF interfaces, and programming language issues pertaining to the use of Fortran-77, ANSI C and K&R C in HDF programming.

2.2 The Hierarchy of HDF Interaction

To review the description of HDF provided in Chapter 1, HDF is a physical file format at its lowest level and a collection of utilities and applications at its highest. Between these two levels, HDF is a library that itself provides two levels of programming interfaces. HDF can be thought of conceptually as three interface layers built upon a physical file format.

FIGURE 2a **The Three Levels of Interaction with the HDF File Format**



Refer to Figure 2a. Of the three types of top-level general applications, only the command-line utilities will be extensively covered in this manual. See Chapter 13, titled *HDF Command-Line Utilities*, for descriptions of this aspect of HDF. These general applications directly call the single-file and multifile interfaces.

The two interactive levels immediately below this level, the ***low-level interface*** and the HDF data file itself, are only briefly described as the single-file and multifile interfaces provide a safer and more standardized means of accessing these levels. The single-file and multifile interfaces - the second highest level of interaction within HDF - are routinely updated as aspects of these lower-level interfaces are changed, in a manner as transparent to the HDF user as possible. With the exception of the few instances where lower-level interface functionality has not yet been incorporated into the higher-level interface functions, the HDF user need not directly concern themselves with these levels.

2.3 Data Objects

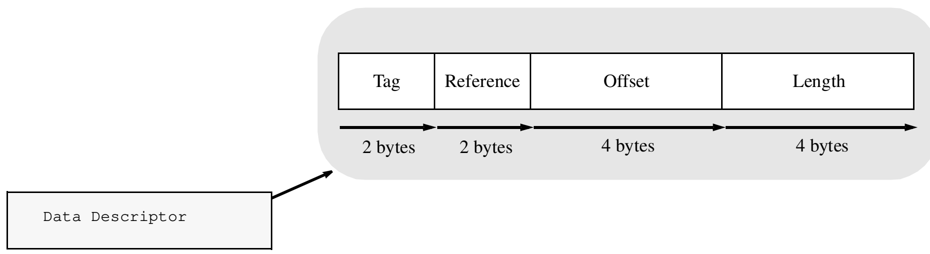
The term ***data object*** is used to describe the fundamental conglomerate structure use to encapsulate data.

Data objects contain a ***data descriptor*** and a ***data element***. Data descriptors consist of information about the type, location, and size of a data element. Data elements contain the primary data itself.

2.3.1 Data Descriptors

All data descriptors are twelve bytes long and contain four fields. (See Figure 2b.)

FIGURE 2b **The Contents of a Data Descriptor**



Tags

Tags identify the type of data stored in its data element. For example, a raster image descriptor is identified by a `DFTAG_RI` tag and a palette descriptor a `DFTAG_LUT` tag. There are currently over 200 tags defined for general use. A complete list of tags and their descriptions can be found in Appendix A of this manual

Tag values ranging between 1 and 32,767 are reserved for commonly-used data types. These tags are assigned by the HDF development group. Tag values ranging between 32,768 and 64,999 are not regulated by NCSA and are available for private application. These tags are not documented by NCSA and may therefore conflict with tags assigned by someone else. Therefore, it is best to limit applications to the use of the commonly-used tags.

Reference Numbers

Reference numbers distinguish between different data elements with the same tag. For example, all raster image descriptors will have the same raster data tag. The pairing of a tag and a reference number provides a unique identifier for any HDF object within a file.

Although HDF assigns reference numbers in increasing order, it is the write operations that are counted, not the number or type of data objects added to the file. For example, writing a raster image set followed by a scientific data set uses a minimum of six data objects but only two write operations. Consequently, HDF will assign 1 as the reference number for the first raster image set and all its members and 2 as the reference number for the scientific data set and all its members.

While application programmers may find it convenient to impart some additional meaning to reference numbers, it should be noted that HDF will not internally recognize any such meaning.

Offsets and Lengths

The offset field points to the location of the data element in the file by storing the number of bytes from the beginning of the file to the beginning of the data element. The length field contains the total size of the data element in bytes.

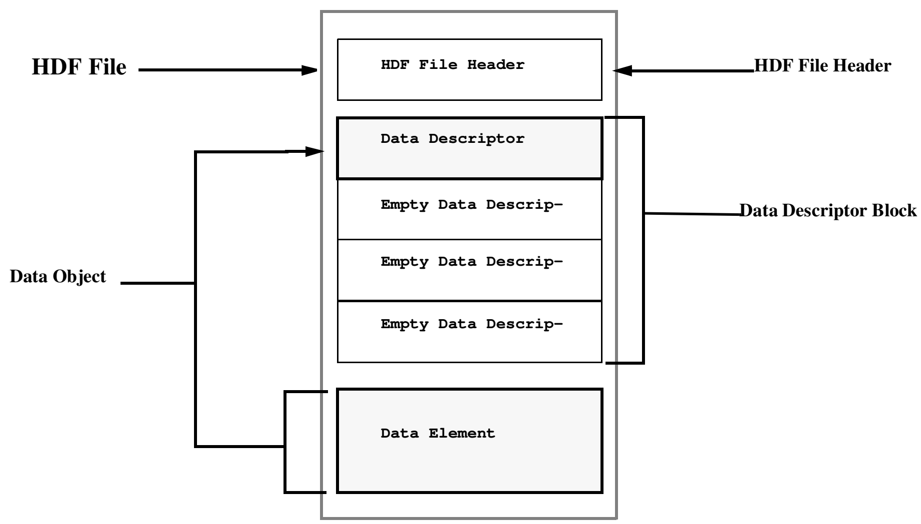
Data Elements

The type of the data stored in a data element is identified by its tag, however, other interpretive information may be required before it can be processed properly.

2.4 File Format

HDF files contain a ***file header*** and at least one ***data descriptor block***, as depicted in Figure 2c. The HDF file header occupies the first four bytes of every HDF file with a signature field containing the 32-bit hexadecimal value `0e031301`. This number is considered a "magic cookie" as it identifies the file as an HDF file. Initially the data descriptor block consists of a group of empty data descriptors. As data objects are written to the file, the HDF library fills the data descriptor block by pairing one data descriptor for each data element. Refer to the following figure.

FIGURE 2c **The Physical Layout of an HDF File Containing One Data Object**



2.4.1 Grouping Data Objects in an HDF File

HDF files that contain more than one data element are generally easier to work with when the data objects containing related data are grouped together. These groups of data objects are called ***data sets***. The HDF user uses the application interface to define, manipulate and dispose of data sets in a file.

As an example, an 8-bit raster image data set requires three objects: a group object identifying the members of the set, an image object containing the image data and a dimension object indicating the size of the image. It is sometimes possible to add additional data objects to the minimum set - for example, an 8-bit raster image set may include a palette.

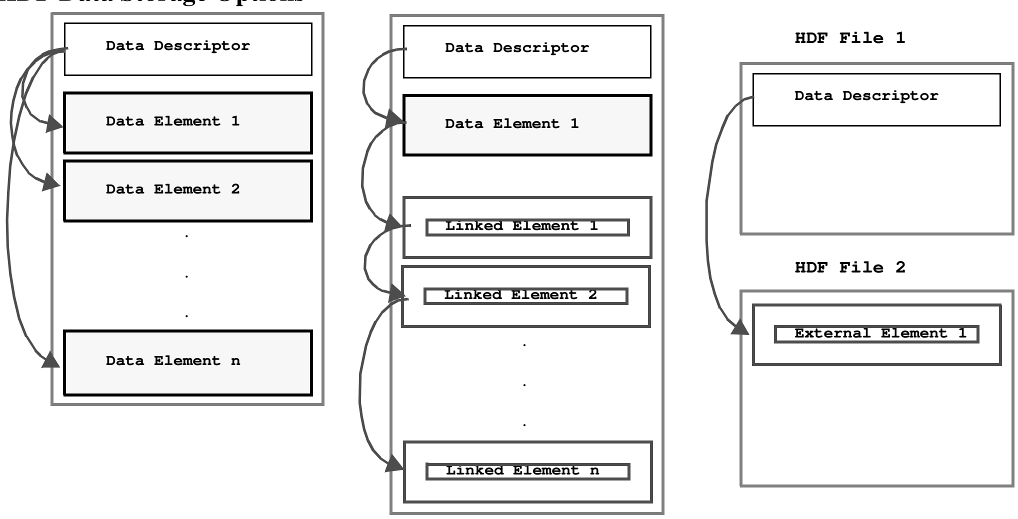
Data objects are individually accessible even if they are included in a set, therefore data objects can belong to more than one set and sets can be included in larger groups. For example, a palette object included in one raster image set may also be a part of another raster image set if its tag and reference number are included in a data descriptor within that second set.

Several data sets may be further grouped into ***group objects***. The contents of the group object depends on the HDF data set it supports.

2.4.2 Storing Data Objects

Data objects can be stored in HDF data files as a ***contiguous element***, as ***linked-block elements*** or as ***external elements***. These data storage options are illustrated below in Figure 2d.

FIGURE 2d **HDF Data Storage Options**



Contiguous Elements Linked-block Elements External Elements

2.4.2.1 Contiguous Data Elements

This is the default method of data storage in HDF. In this case, all data objects exist in one HDF file and are stored in the manner illustrated in Figure 2c. Each data element in the file is one complete unit of raw data, and as only one unique data descriptor points to it there are no references or relationships to the other data elements in the file. As additional data elements are created and written they are appended to the end of the HDF file.

2.4.2.2 Linked-block Data Elements

Linked-block elements are a series of data elements existing in one HDF file which serve as a means of adding data to a pre-existing data element. These elements are linked to each other and to the original data element by a linked-list structure similar to the data descriptor list. They are of a uniform size with the exception of the first block after the original data element, which is the only block that can be resized.

2.4.2.3 External Data Elements

External data elements are those that exist in a file apart from the one the data descriptor list resides in. These additional HDF data files are referred to as ***external data files***. The HDF library keeps track of the filesystem location of the external data file as well as the standard tag/reference number pair, offset and length information of the external element.

2.5 Header File Information

The “hdf.h” header file must be included in every HDF application program as it contains necessary declarations and definitions as well as prototypes for the HDF API routines. To use these routines, the HDF user must be familiar with the most-commonly used definitions stored in the “hdf.h” file.

2.5.1 File Access Code Definitions

These definitions are passed into API file access routines as parameters. The `DFACC_RDONLY` and `DFACC_CLOBBER` definitions exist in the “hdf.h” file to accommodate backward compatibility with applications designed to work with older versions of the HDF API library.

TABLE 2A **File Access Code Definitions**

Definition Name	Definition Value	Description
DFACC_READ	1	Read access.
DFACC_WRITE	2	Write access.
DFACC_RDWR	3	Read and write access.
DFACC_CREATE	4	File creation access.
DFACC_CLOBBER	4	Same as DFACC_CREATE.
DFACC_RDONLY	1	Same as DFACC_READ.

2.5.2 Data Type Definitions

These definitions are used in comparison expressions to determine the type of an input variable or a value returned by an API function. `DFNT_FLOAT`, `DFNT_DOUBLE`, `DFNT_UCHAR`, and `DFNT_CHAR` are included in the “hdf.h” file for backward compatibility. These definitions are located in the “hntdefs.h” header file.

TABLE 2B **Data Type Definitions**

Definition Name	Definition Value	Description
DFNT_CHAR8	4	8-bit character type.
DFNT_CHAR	4	Same as DFNT_CHAR8.
DFNT_UCHAR8	3	8-bit unsigned character type.
DFNT_UCHAR	3	Same as DFNT_UCHAR8.
DFNT_INT8	20	8-bit integer type.
DFNT_UINT8	21	8-bit unsigned integer type.
DFNT_INT16	22	16-bit integer type.
DFNT_UINT16	23	16-bit unsigned integer type.
DFNT_INT32	24	32-bit integer type.
DFNT_UINT32	25	32-bit unsigned integer type.
DFNT_INT64	26	64-bit integer type.
DFNT_UINT64	27	64-bit unsigned integer type.

DFNT_FLOAT32	5	32-bit floating-point type.
DFNT_FLOAT64	6	64-bit floating-point type.

2.5.3 Tag Definitions

These definitions identify the object tags defined and used by the HDF API library. The concept of object tags is introduced in Section 2.3.1 on page 8. Note that, in the tag list that follows in Table 2C, tags can also identify properties of data objects such as raster image compression and pixel interlacing type (these concepts are described in Chapter 6, titled *8-bit Raster Images. (DFR8 API)* 8- and 24-raster images and scientific data sets are respectively described in Chapter 6, titled *8-bit Raster Images (DFR8 API)*, Chapter 7, titled *24-bit Raster Images (DF24 API)* and Chapter 3, titled *Scientific Data Sets (SD API)*.

TABLE 2C **Tag Definitions**

Definition Name	Definition Value	Description
DFTAG_FID	((uint16) 100)	File identifier.
DFTAG_FD	((uint16) 101)	File description.
DFTAG_TID	((uint16) 102)	Tag identifier.
DFTAG_TD	((uint16) 103)	Tag descriptor.
DFTAG_DIL	((uint16) 104)	Data identifier label.
DFTAG_DIA	((uint16) 105)	Data identifier annotation.
DFTAG_NT	((uint16) 106)	Number type.
DFTAG_ID8	((uint16) 200)	8-bit raster image dimension record.
DFTAG_IP8	((uint16) 201)	8-bit raster image palette.
DFTAG_RI8	((uint16) 202)	8-bit raster image data.
DFTAG_CI8	((uint16) 203)	8-bit raster image data - RLE compressed.
DFTAG_II8	((uint16) 204)	8-bit raster image data - IMCOMP compressed.
DFTAG_ID	((uint16) 300)	24-bit raster image dimension record.
DFTAG_LUT	((uint16) 301)	24-bit raster image palette.
DFTAG_RI	((uint16) 302)	24-bit raster image data.
DFTAG_CI	((uint16) 303)	24-bit raster image - compressed.
DFTAG_RIG	((uint16) 306)	Raster image group.
DFTAG_LD	((uint16) 307)	Palette dimension record.
DFTAG_SDG	((uint16) 700)	Scientific data group.
DFTAG_SDD	((uint16) 701)	Scientific data dimension record.
DFTAG_SD	((uint16) 702)	Scientific data group data.
DFTAG_SDS	((uint16) 703)	Scientific data scale.
DFTAG_SDL	((uint16) 704)	Scientific data label.
DFTAG_SDU	((uint16) 705)	Scientific data unit.
DFTAG_NDG	((uint16) 720)	Numeric data group.
DFTAG_CAL	((uint16) 731)	Calibration information.
DFTAG_VG	((uint16) 1965)	Vgroup.
DFTAG_VH	((uint16) 1962)	Vdata header.
DFTAG_VS	((uint16) 1963)	Vdata storage.
DFTAG_RLE	((uint16) 11)	Run-length encoding compression algorithm.
DFTAG_IMCOMP	((uint16) 12)	IMCOMP compression algorithm.
DFTAG_JPEG	((uint16) 13)	JPEG compression algorithm - 24-bit data.
DFTAG_GREYJPEG	((uint16) 14)	JPEG compression algorithm - 8-bit data.

DFIL_PIXEL	((uint16) 0)	Pixel interlacing.
DFIL_LINE	((uint16) 1)	Scan-line interlacing.
DFIL_PLANE	((uint16) 2)	Scan-plane interlacing.

2.5.4 Limit Definitions

These definitions declare the maximum size of specific data object parameters, such as the maximum length of a vdata field or the maximum number of objects in a vgroup. Vdata objects are discussed in Chapter 4, titled *Vdatas (VS API)*, and vgroup objects are described in Chapter 5, titled *Vgroups (V API)*. Except for `FIELDNAMELENMAX`, these can be safely altered by the HDF user.

TABLE 2D **Limit Definitions**

Definition Name	Definition Value	Description
FIELDNAMELENMAX	128	Maximum length of a vdata field in bits - 16 characters.
VSNAMELENMAX	64	Maximum length of a vdata name in bytes - 64 characters.
VGNAMELENMAX	64	Maximum length of a vgroup name in bytes - 64 characters.
MAX_VFILE	16	Maximum number of open data files.
MAXNVELT	64	Maximum number of objects in a vgroup.
MAX_ORDER	32000	Maximum order of a vdata field.
MAX_FIELD_SIZE	32000	Maximum length of a field.

2.6 Basic Operations on HDF Files

The HDF programming model specifies that a data file be first explicitly opened by an application, then manipulated, then explicitly closed by the application code at the end of its execution. To use the routines designed to open and close data files, the user must first know about the ***file identifiers*** used by the HDF API routines.

2.6.1 File Identifiers

HDF data files are uniquely identified by either a filename or a file identification number, or ***file id***. The filename is the name of the file as represented in the native filesystem, and is created along with the file itself through the HDF file creation routine. The file id is the numeric identifier given to the file by the HDF library, also at the time of creation, and is generally only used by the HDF library routines; the HDF user need not keep track of them.

As every file is assigned its own identifier, the order in which files are accessed is very flexible. For example, it is perfectly valid to open a file and obtain an identifier for it, then open a second file without closing the first file or disposing of the first file identifier. The only requirement made by HDF is that all file ids be individually discarded before the termination of the calling program.

File identifiers created by any HDF API routine cannot be used by the routines of any other interface - they are not interchangeable.

2.6.2 Opening HDF Files: Hopen

The **Hopen** routine opens or creates an HDF data file, depending on the access mode specified, and returns the file id the HDF library has assigned it. The parameter names and data types are listed in Table 2E below. Refer also to the *HDF Reference Manual* for additional information on **Hopen** and to Section 2.5 on page 11 for information regarding file access codes.

TABLE 2E **Hopen Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type C	Data Type Fortran-77	Description
Hopen **(hopen)**	filename	char *	character8 (*)	File identifier.
	access	intn	integer	Access mode definition.
	n_dds	int16	integer*2	Number of data descriptors in a block if a new file is to be created.

2.6.3 Closing HDF Files: Hclose

The **Hclose** routine closes the file designated by the file id passed in as the `file_id` parameter. The parameter names and data types are listed in Table 2F below. Refer also to the *HDF Reference Manual* for additional information regarding **Hclose**.

TABLE 2F **Hclose Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type C	Data Type Fortran-77	Description
Hclose **(hclose)**	file_id	int32	integer*4	File identifier.

2.6.4 Determining the Number of Objects with a Specified Tag: Hnumber

Hnumber determines how many objects with the specified tag exist within a file. To count the total number of objects in a file, set the `tag` argument to `DFTAG_WILDCARD`. Note that a return value of 0 is not a error condition. The parameter names and data types are listed in Table 2G below. Refer also to the *HDF Reference Manual* for additional information regarding **Hnumber**.

TABLE 2G **Hnumber Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type C	Data Type Fortran-77	Description
Hnumber **(hnumber)**	file_id	int32	integer*4	File identifier.
	tag	int32	uint16	Tag to be counted.

2.6.5 Getting the HDF Library Version Used to Create a File: Hgetlibversion

Hgetlibversion returns the version of the HDF library currently being used, as well as additional textual information regarding the library. The parameter names and data types are listed in Table 2H below. Refer also to the *HDF Reference Manual* for additional information regarding **Hgetlibversion**.

TABLE 2H

Hgetlibversion Parameter List

Function Name (Fortran-77)	Parameter	Data Type C	Description
Hgetfileversion	major_v	`uint32 *`	Major version number.
	minor_v	`uint32 *`	Minor version number.
	release	`uint32 *`	Complete library version number.
	string	`char [80]`	Additional information about the library version.

2.6.6 Locate an Object by its Tag/Reference Number Pair: Hfind

Hfind, like all H functions, must be preceded by a call to **Hopen** and followed at some point by a call to **Hclose**.

Although **Hfind** is capable of executing many kinds of search operations, it is particularly useful for determining the valid reference numbers for any specified tag. When supplied with a recognized HDF tag, a wildcarded reference number and a search direction, **Hfind** will search sequentially through the objects stored in an HDF file until the first data object with the specified tag is encountered. Once the data object is discovered, **Hfind** will return its reference number. By using **Hfind** in a conditional loop the reference number for any or all data objects in an HDF file can be retrieved. For more about **Hfind** consult the *HDF Reference Guide*.

TABLE 2I

Hfind Parameter List

Routine Name (Fortran-77)	Parameter	Data Type C	Description
Hfind	major_v	`uint32 *`	Major version number.
	minor_v	`uint32 *`	Minor version number.
	release	`uint32 *`	Complete library version number.
	string	`char [80]`	Additional information about the library version.

2.7 Application Programming Interfaces

HDF provides Fortran-77 and C APIs for storing and retrieving 8- and 24-bit raster images, palettes, scientific data, and annotations. These interfaces are described in detail in Chapters 3 through 12 of this manual.

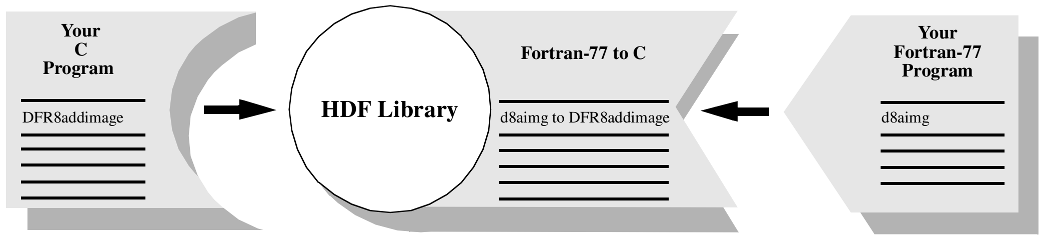
2.8 Fortran-77 and C Language Issues

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.

2.8.1 Fortran-77-to-C Translation

Nearly all of the HDF library code is written in C. The Fortran-77 HDF API routines translate all parameter data types to C data types, then call the C routine that performs the main function. For example, **d8aimg** is the Fortran-77 equivalent for **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.

FIGURE 2e **Use of a Function Call Converter to Route Fortran-77 HDF Calls to the C Library**



2.8.2 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.

2.8.3 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.

2.8.4 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 `m4` and `cpp` let your compiler include and preprocess header files. If this capability is not available, you may have to copy whatever declarations, definitions, or values you need from the “constants.f” file into your program code. If it is, include the header file named “hdf.inc” in your Fortran-77 code. The “constants.f” file is included in the “hdf.inc” header file.

2.8.5 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 2J.)

TABLE 2J **Data Type Definitions**

Data Type	C	Fortran-77
8-bit signed integer	`int8`	`integer*1`
8-bit unsigned integer	`uint8`	`character*1`
16-bit signed integer	`int16`	`integer*2`
16-bit unsigned integer	`uint16`	Not supported.
32-bit signed integer	`int32`	`integer*4`
32-bit unsigned integer	`uint32`	Not supported.

Data Type	C	Fortran-77
32-bit floating point number	`float32`	`real*4`
64-bit floating point number	`float64`	`real*8`
Native signed integer	`intn`	`integer`
Native unsigned integer	`uintn`	Not supported.

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.

2.8.6 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 an array that contains an indefinite number of characters. It is the responsibility of the calling program to allocate enough space to hold whatever data is stored in the array.
- `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.

2.8.7 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.

2.9 Low-Level Interfaces

The low-level HDF interface consists of the ***H interface***, the ***HL/HX interface*** and the ***HE interface***.

2.9.1 The H Interface

The low-level H interface builds and manipulates data objects in an HDF file. A thorough explanation of each H interface routine can be found in the *HDF Specification and Developer's Guide*. Table 2K lists and briefly describes the H interface routines that are most commonly used in HDF applications.

TABLE 2K **Some H Interface Routines**

Category	Routine Name	Description
Input/Output	`Hopen`	Provides an access path to a file.
	`Hclose`	Closes an access path to a file.
	`Hgetlibversion`	Returns version information for the current HDF library.

2.9.2 The HX Interface

The HX interface routines create and maintain linked and external data elements. A thorough explanation of each HX interface routine can be found in the *HDF Specification and Developer's Guide*.

2.9.3 The HE Interface

The HE interface routines provide error handling functionality. The HE interface routines are described in the *HDF Specification and Developer's Guide* and Chapter 12, titled *Error Reporting*.

Chapter 3

Scientific Data Sets (SD API)

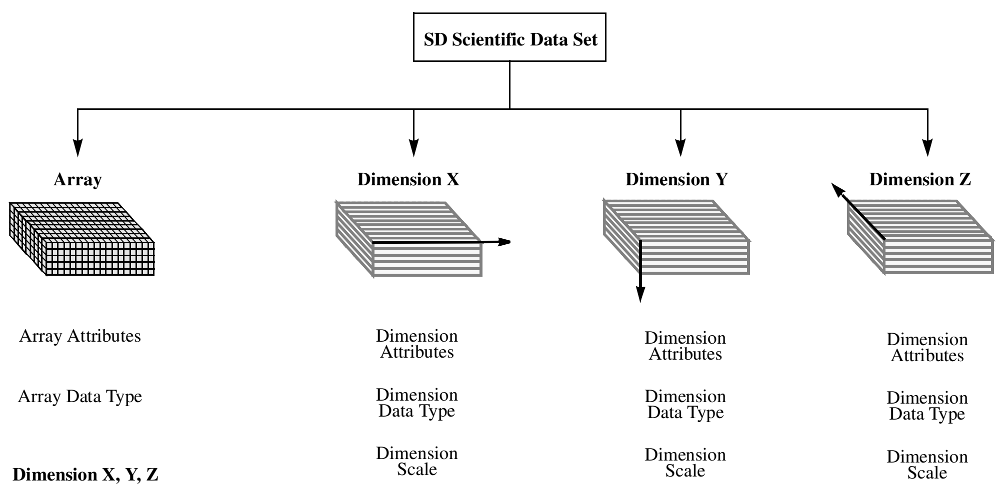
3.1 Chapter Overview

This chapter describes the routines available for storing and retrieving multidimensional arrays containing scientific data.

3.2 The SD Scientific Data Set Data Model

In HDF, any multi-dimensional array qualifies as a ***scientific data set*** or ***SDS*** if it's associated with a dimension record and a data type. In addition to providing a framework for storing arrays of arbitrary dimensions and data type, the SDS data model supports ***dimension scales***, ***user-defined attributes*** and ***predefined attributes***. (See Figure 3a.)

FIGURE 3a **The Contents of a Three-Dimensional SD Scientific Data Set**



Scientific data sets consist of required and optional objects. The required objects will be covered first. Please note that in this chapter the terms "SDS" and "data set" are used interchangeably.

3.2.1 Required SD SDS Objects

Every SD scientific data set must contain three objects. These objects include the ***SDS array***, ***data type*** and ***dimension records***. Required objects are automatically created from the information provided at the time the SDS is defined.

3.2.1.1 SDS Array

An ***SDS array*** is an n-dimensional data structure that serves as the basic building block of an SDS. When an SDS array is created, the number and size of the dimensions that define its shape are specified, as is its data type. SDS arrays are conceptually equivalent to ***variables*** in the netCDF data model.

3.2.1.2 SDS Array Name

An SDS array has an ***SDS name*** consisting of a sequence of case-sensitive alphanumeric characters. A name can be assigned to an SDS by a calling program, but if a name is not provided by the calling program one will be assigned by the HDF library. Names are assigned when the data set is created and cannot be changed. SDS names do not have to be unique within a file, but if they are not it can be difficult to distinguish between the scientific data sets in the file.

3.2.1.3 Data Type

The standard data types supported by the SD API are 32- and 64-bit floating-point numbers, 8-, 16- and 32-bit signed integers and 8-, 16- and 32-bit unsigned integers. The SD interface also includes a routine that allows SD data sets with variable bit lengths (1 to 32 bits) to be created.

Before writing an SDS to a file, HDF normally converts its elements from the native format of the host machine to a standard HDF format. The standard representations used by HDF for floating-point numbers are the IEEE 32- and 64-bit floating-point formats. For integers, HDF uses big-endian byte ordering. For signed integers HDF uses twos-complement representation. Converting to and from the standard formats can result in low-order inaccuracies in the data. For example, data converted from 64-bit to 32-bit floating-point representation is accurate to about 10^{-7}.

Sometimes users prefer not to have their data automatically converted, either because the conversion slows down processing or because it introduces intolerable inaccuracies. For those instances, HDF provides a “native format” option, whereby numbers are stored "as is" in the file and are tagged accordingly. HDF also provides a “little-endian” option to suppress any rearranging of byte ordering from little- to big-endian. This is primarily for users of Intel-based machines who do not want to incur the cost of reordering data when writing to an HDF file. Because HDF does not support direct conversion between many machine architectures, using a native format can diminish the portability of HDF files. However, note that direct conversions are supported between little-endian and all other byte-order formats supported by HDF.

3.2.1.4 Dimensions

SDS ***dimensions*** specify the shape and size of an SDS array. The number of dimensions of an array is known as the ***rank*** of the array. Dimension names are not treated in the same way as array names. For example, if a name assigned to a dimension was previously assigned to another dimension the SD interface treats both dimensions as the same data object and any changes made to one will be reflected in the other. The size of a dimension is a positive integer.

Also, one dimension of an SDS array can be assigned the predefined size `SD_UNLIMITED`. This dimension is referred to as an ***unlimited dimension*** - which, as the name suggests, can grow to any length.

3.2.2 Optional SD SDS Objects

There are two types of optional objects available for inclusion in an SDS: ***dimension scales*** and ***attributes***. Attributes are either predefined or defined by the user. Optional objects are only created when specifically requested by the calling program.

Dimension Scales

A dimension scale is a sequence of numbers placed along a dimension to demarcate intervals along it. Dimension scales are described in Section 3.9 on page 71.

User-Defined Attributes

Attributes are alphanumeric strings describing the nature and/or intended usage of the file, SDS or dimension they're attached to. ***User-defined attributes*** are attributes defined by the calling program containing auxiliary information about a file, SDS array or dimension. They are more fully described in Section 3.10 on page 80.

Predefined Attributes

Predefined attributes are attributes that have reserved labels and in some cases predefined data types. Predefined attributes are useful because they establish conventions that applications can depend on. They are further described in Section 3.11 on page 87.

3.2.3 Annotations and the SD Data Model

With the expansion of the SD interface to include user-defined attributes, annotations should no longer be used in conjunction with scientific data sets. In fact, metadata once stored as an annotation is now more conveniently stored as an attribute. However, to insure backward compatibility with scientific data sets and applications relying on annotations, the DFAN annotation API, described in Chapter 10, titled *Annotations (DFAN API)*, can be used to annotate SDSs. There is no cross-compatibility between attributes and annotations; creating one does not automatically create the other.

3.3 The SD Scientific Data Set API

Unlike the DFSD SDS interface, the SD interface supports simultaneous access to more than one SDS in more than one HDF file. Although it is fully compatible with the DFSD data model, the SD interface also supports a more powerful and more general scientific data model; one that is very similar to the netCDF data model developed by the Unidata Program Center.

All valid operations involving an SDS can be done by calling routines in the SD interface. Except in rare instances, the physical file format used to store HDF data is transparent to HDF users.

3.3.1 SD Library Routines

All C routines in the SD scientific data library begin with the prefix "SD". The equivalent Fortran-77 functions use the prefix "sf". These routines are categorized as follows:

- ***Access routines*** initialize and terminate access to HDF files and data sets.
- ***Read and write routines*** read and write data sets by manipulating their dimensions, rank and data type.
- ***General inquiry routines*** return information about the location, contents and description of the scientific data sets in an HDF file.
- ***Dimension scale routines*** define and access dimension scales within a data set.
- ***User-defined attribute routines*** describe and access characteristics of an HDF file, data set or dimension defined by the HDF user.
- ***Predefined attribute routines*** access previously-defined characteristics of an HDF file, data set, or dimension.

- ***Compression routines*** determine the compression method for SDS data.
- ***Chunking/tiling routines*** determine the chunking configuration for SDS data.
- ***N-bit data set routines*** determine the non-standard data bit length configuration for SDS data.

The SD routines are described in the following table and in the *HDF Reference Guide*.

TABLE 3A

SD Library Routines

Category	Routine Name		Description
	C	Fortran-77	
Access	`SDend`	`sfend`	Closes the file and clean up memory.
	`SDendaccess`	`sfendacc`	Disposes of a data set identifier, flush out metadata and order information.
	`SDselect`	`sfselect`	Returns the identifier of the specified data set.
	`SDstart`	`sfstart`	Initializes the SD interface.
Read and Write	`SDcreate`	`sfcreate`	Creates a new data set.
	`SDreaddata`	`sfrdata/` `sfrcdata`	Reads data from a chunked or non-chunked data set.
	`SDsetexternalfile`	`sfsextf`	Defines the data type to be stored in an external file.
	`SDwritedata`	`sfwdata/` `sfwcdata`	Writes data to a chunked or non-chunked data set.
General Inquiry	`SDfileinfo`	`sffinfo`	Returns information about the contents of a file.
	`SDgetinfo`	`sfginfo`	Returns information about a data set.
	`SDidtoref`	`sfid2ref`	Returns a reference number for a named data set.
	`SDiscoordvar`	`sfiscvar`	Distinguishes data sets from dimension scales.
	`SDnametoindex`	`sfn2index`	Returns an index of a specified data set.
	`SDreftoindex`	`sfref2index`	Returns the index of a data set corresponding to a given reference number.
Dimension Scales	`SDdiminfo`	`sfgdinfo`	Gets information about a dimension.
	`SDgetdimid`	`sfdimid`	Retrieves the identifier of a dimension.
	`SDsetdimname`	`sfsdimname`	Associates a name with a dimension.
	`SDgetdimscale`	`sfgdscale`	Returns scale values for a dimension.
	`SDsetdimscale`	`sfsdscale`	Defines the values of this dimension.
User-defined Attributes	`SDattrinfo`	`sfgainfo`	Gets information about an attribute.
	`SDfindattr`	`sffattr`	Returns the index of the specified attribute.
	`SDreadattr`	`sfrnatt/` `sfrcatt`	Reads the values of the specified attribute.
	`SDsetattr`	`sfsnatt/` `sfscatt`	Creates and defines a new attribute.

Predefined Attributes	SDgetcal	sfgcal	Returns calibration information.
	SDgetdatastrs	sfgdtstr	Returns the label, limit, format and coordinate system of a data set.
	SDgetdimstrs	sfgdmstr	Returns the attribute strings for a dimension.
	SDgetfillvalue	sfgfill/ sfgcfill	Reads the fill value if it exists.
	SDgetrange	sfgrange	Returns the range of values of the specified data set.
	SDsetcal	sfscal	Defines the calibration information.
	SDsetdatastrs	sfsdtstr	Defines the attribute strings of the specified data set.
	SDsetdimstrs	sfsdmstr	Defines the attribute strings of the specified dimension.
	SDsetfillvalue	sfsfill/ sfscfill	Defines the fill value of the current data set.
	SDsetfillmode	sfsflmd	Defines the file mode to be applied to all SDSs in the specified file.
	SDsetrange	sfsrange	Defines the maximum and minimum values of the valid range.
Compression	SDsetcompress	None	Defines the compression method to be applied to data set data.
Chunking/ Tiling	SDgetchunkinfo	None	Obtains information about a chunked SDS.
	SDsetchunkcache	None	Sets the size of the chunk cache.
	SDsetchunk	None	Makes a non-chunked SDS a chunked SDS.
	SDwritechunk	None	Writes data to a chunked SDS.
	SDreadchunk	None	Reads data from a chunked SDS.
N-bit Data Length	SDsetnbitdataset	sfsnbit	Defines the non-standard bit length of the data set data.

3.3.2 SDS Identifiers in the SD Interface

In the SD interface, scientific data sets are identified in several ways. Before a data set is accessible, it is identified by ***index***, ***name*** and ***reference number***. After it is opened, it is identified by a ***data set identifier*** or ***SDS id***.

The index describes the relative position of the data set in the file. The name is a unique character string and the reference number is a unique integer which are both assigned to the data set when it is created. The index, name, and reference number are needed to obtain a data set id.

The SDS id is the data set equivalent of a file identifier and uniquely identifies an SDS data set within a file. It is created when an existing SDS is selected for use or when a new SDS is created and is thereafter used to access the SDS.

3.3.3 Tags in the SD Interface

A complete list of SDS tags and their descriptions is in Table D in the User's Guide Appendix.

3.4 Programming Model for the SD Interface

To support multifile access, the SD interface relies on the calling program to initiate and terminate access to files and data sets. The SD programming model for accessing an SDS in an HDF file is as follows:

1. Open a file and initialize the SD interface.
2. Open an existing SDS by obtaining an SDS id from an SDS index **OR** create a new SDS by obtaining an SDS id from the SDS name, rank and dimensions.
3. Perform desired operations on the SDS.
4. Terminate access to the data set.
5. Terminate access to the SD interface and close the file.

To access a single SDS in an HDF file, the calling program must contain the following calls:

C:
```
sd_id = SDstart(filename, access_mode);
sds_id = SDselect(sd_id, sds_index);
<Optional operations>
status = SDendaccess(sds_id);
status = SDend(sd_id);
```

FORTRAN:
```
sd_id = sfstart(filename_1, access_mode)
sds_id = sfselect(sd_id, sds_index)
<Optional operations>
status = sfendaccess(sds_id)
status = sfend(sd_id)
```

To access several files at the same time, a calling program must obtain a separate file id for each file to be opened. Likewise, to access more than one SDS a calling program must obtain a separate SDS id for each SDS. For example, to open two SDSs stored in two files a program would execute a series of function calls similar to the following:

C:
```
sd_id_1 = SDstart(filename_1, access_mode);
sds_id_1 = SDselect(sd_id_1, sds_index_1);
sd_id_2 = SDstart(filename_2, access_mode);
sds_id_2 = SDselect(sd_id_2, sds_index_2);
<Optional operations>
status = SDendaccess(sds_id_1);
status = SDend(sd_id_1);
status = SDendaccess(sds_id_2);
status = SDend(sd_id_2);
```

FORTRAN:
```
sd_id_1 = sfstart(filename_1, access_mode)
sds_id_1 = sfselect(sd_id_1, sds_index_1)
sd_id_2 = sfstart(filename_2, access_mode)
sds_id_2 = sfselect(sd_id_2, sds_index_2)
<Optional operations>
status = sfendacc(sds_id_1)
status = sfend(sd_id_1)
status = sfendacc(sds_id_2)
status = sfend(sd_id_2)
```

As with file identifiers, SD ids can be obtained and discarded in any order and all SD ids must be individually discarded before termination of the calling program.

3.4.1 Establishing Access to Files and Data Sets: SDstart and SDselect

In the SD interface, **SDstart** is used to open files rather than **Hopen**. **SDstart** takes two arguments; `filename` and `access_mode`, and returns the file id `sd_id`. The argument `filename` is the name of an HDF or netCDF file as it is stored on disk. All other functions in the SD interface accept only `sd_id` for file operations. The argument `access_mode` specifies the type of access required for operations on the file. The access mode tags passed in the `access_mode` parameter have names prefaced by "DFACC".

Although it is possible to open a file more than once, it is better to select the most appropriate access mode and call **SDstart** only once. Repeatedly calling **SDstart** may cause unexpected results and is not recommended.

If `access_mode` is set to `DFACC_RDONLY`, the specified file will not be created if it doesn't exist.

SDselect takes two arguments: `sd_id` and `sds_index` and returns the data set id `sds_id`. The argument `sd_id` is the file identifier returned by **SDstart**, and `sds_index` is the position of the data set relative to the beginning of the file. The argument `sds_index` is zero-based, meaning that the index of first SDS in the file is 0.

The parameters of **SDstart** and **SDselect** are further defined below. (See Table 3B.)

3.4.2 Terminating Access to Files and Data Sets: SDendaccess and SDend

SDendaccess disposes of the open data set id `sds_id` and terminates access to the data set. The calling program must make one **SDendaccess** call for every **SDselect** call made during its execution. Failing to call **SDendaccess** for each call to **SDselect** or **SDcreate** may result in a loss of data.

SDend disposes of the file id `file_id` and terminates access to the file and the SD interface. The calling program must make one **SDend** call for every **SDstart** call made during its execution. Failing to call **SDend** for each **SDstart** may result in a loss of data.

The parameters of **SDendaccess** and **SDend** are further defined in the following table.

TABLE 3B **SDstart, SDselect, SDend and SDendaccess Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDstart (sfstart)	filename	char *	character* (*)	Name of the HDF or netCDF file.
	access_mode	int32	integer	Type of access.
SDselect (sfselect)	file_id	int32	integer	HDF file identifier.
	sds_index	int32	integer	Position of the data set within the file.
SDend (sfend)	file_id	int32	integer	HDF file identifier.
SDendaccess (sfendacc)	sds_id	int32	integer	Data set identifier.

EXAMPLE 1. **Accessing and Closing an SDS**

The following examples are a code template on how to access the first data set in an HDF file, then detach from the data set and the file. The file name "Dummy_HDF_File.hdf" represents a preexisting HDF file - replace it with the name of your target file.

This example assumes that the "Dummy_HDF_File.hdf" file contains one SDS.

C:

```
#include "mfhdf.h"

main( )
{
int32 sd_id, sds_id, sds_index, status;
int32 rank, num_type, attributes;

/* Open the HDF file. DFACC_CREATE is defined in hdf.h. */
sd_id = SDstart("Dummy_HDF_File.hdf", DFACC_CREATE);

/* Get the identifier of the first data set. */
sds_index = 0;
sds_id = SDselect(sd_id, sds_index);
```

```
/* Dispose of the data set identifier to terminate access. */
status = SDendaccess(sds_id);

/* Dispose of the file identifier to close the file. */
status = SDend(sd_id);

}
```

FORTRAN:

```
      PROGRAM SDSDATA ACCESS

      integer*4 sds_id, sd_id, sds_index, status
      integer sfstart, sfselect, sfendacc, sfend

C     DFACC_CREATE is defined in hdf.h.
      parameter (DFACC_CREATE = 4)

C     Open the HDF file.
      sd_id = sfstart('Dummy_HDF_File.hdf', DFACC_CREATE)

C     Get the identifier of the first data set.
      sds_index = 0
      sds_id = sfselect(sd_id, sds_index)

C     Dispose of the data set identifier to terminate access.
      status = sfendacc(sds_id)

C     Dispose of the file identifier to close the file.
      status = sfend(sd_id)

      end
```

3.5 Creating and Writing to Simple Scientific Data Sets

This section describes the routines needed to create and write simple SDSs. A “simple” SDS is defined here as one with no attributes or user-defined dimension scales.

In the SD interface, creating and writing data to an SDS are separate operations - each performed by one SD routine. The SD interface retains no definitions about the size, contents or rank of an SDS from one SDS to the next or from one file to the next.

3.5.1 Creating Scientific Data Sets: SDcreate

Creating a simple SDS or an SDS without attributes or scales involves the following steps:

1. Open a file and initialize the SD interface.
2. Define the characteristics of the SDS.
3. Perform optional operations on the SDS.
4. Terminate access to the data set.
5. Terminate access to the SD interface and close the file.

To create an SDS, the calling program must contain the following sequence of routine calls:

C:

```
sd_id = SDstart(filename, access_mode);
sds_id = SDcreate(sd_id, name, number_type, rank, dim_sizes);
```

```
<Optional operations>
status = SDendaccess(sds_id);
status = SDend(sd_id);
```

FORTRAN:
```
sd_id = sfstart(filename, access_mode)
sds_id = sfcreate(sd_id, name, number_type, rank, dim_sizes)
<Optional operations>
status = sfendacc(sds_id)
status = sfend(sd_id)
```

SDcreate defines a new SDS using the arguments `name`, `number_type`, `rank` and `dim_sizes`. Once a data set is created, you cannot change its name, data type or rank. **SDcreate** does not actually perform the write; it occurs only when **SDend** is called.

The SD interface will assign a name if one is not provided. In this situation, the `filename` parameter must be a null character string. The maximum length of an SDS name is defined by `MAX_NC_NAME` and the maximum rank of an SDS array is defined by `MAX_VAR_DIMS`. Both are defined in the "netcdf.h" header file. Tag names passed in the `number_type` parameter are prefaced by "DFNT".

When creating an SDS, it is necessary to specify the data type of the array data contained in the SDS. The "hntdefs.h" header file contains definitions of all valid data types, which are described in Chapter 2, titled *HDF Fundamentals*.

TABLE 3C

SDcreate Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDcreate (sfcreate)	file_id	int32	integer	File identifier.
	name	char *	character* (*)	ASCII string containing the name of the data set.
	data_type	int32	integer	Data type of the data set.
	rank	int32	integer	Number of dimensions in the array.
	dim_sizes	int32 []	integer (*)	Array defining the size of each dimension.

EXAMPLE 2.

Creating an Empty SDS

If **SDcreate** is called but not written, an empty array is created. An "empty array" is an array that has been defined but not yet initialized with data. All array information passed into **SDcreate** through its parameters are stored in the file.

C:

```
#include "hdf.h"
#include "mfhdf.h"

#define X_LENGTH 5
#define Y_LENGTH 16

main( )
{

int32 sd_id, sds_id, status;
int32 dimsizes[2], rank;

/* Create and open the file and initiate the SD interface. */
```

```
sd_id = SDstart("Example2.hdf", DFACC_CREATE);

/* Define the rank and dimensions of the array to be created. */
rank = 2;
dimsizes[0] = Y_LENGTH;
dimsizes[1] = X_LENGTH;

/* Create the array. */
sds_id = SDcreate(sd_id, "Ex_array_1", DFNT_INT16, rank, dimsizes);

/* Terminate access to the array. */
status = SDendaccess(sds_id);

/* Terminate access to the SD interface and close the file */
status = SDend(sd_id);

}
```

FORTRAN:

```
     PROGRAM EMPTY ARRAY

      integer*4 sd_id, sds_id, dimsizes(2), rank
      integer sfstart, sfcreate, sfendacc, sfend
      integer*4 X_LENGTH, Y_LENGTH
      parameter (X_LENGTH = 16, Y_LENGTH = 5)

C     DFACC_CREATE and DFNT_INT16 are defined in hdf.h.
      integer*4 DFACC_CREATE, DFNT_INT16
      parameter (DFACC_CREATE = 4, DFNT_INT16 = 22)

C     Create and open the file and initiate the SD interface.
      sd_id = sfstart('Example2.hdf', DFACC_CREATE)

C     Define the rank and dimensions of the array to be created.
      rank = 2
      dimsizes(1) = Y_LENGTH
      dimsizes(2) = X_LENGTH

C     Create the array.
      sds_id = sfcreate(sd_id, 'Ex_array_1', DFNT_INT16, rank,
     +                  dimsizes)

C     Terminate access to the array.
      status = sfendacc(sds_id)

C     Terminate access to the SD interface and close the file.
      status = sfend(sd_id)

      end
```

3.5.2 Writing Data to an SDS Array: SDwritedata

SDwritedata is the only routine that is needed to write data to an SDS array. It can completely or partially fill an SDS n-dimensional array or append data along one dimension defined to be of unlimited length. It can also skip a specified number of SDS array elements between write operations along each dimension.

Creating an SDS and writing one or more slabs to it involves the following steps:

1. Create an SDS.

2. Write a slab or series of slabs.

To do this, the calling program must contain the following sequence of routine calls:

C:
```
sds_id = SDcreate(sd_id, name, number_type, rank, dim_sizes);
status = SDwritedata(sds_id, start, stride, edge, data);
```

FORTRAN:
```
sds_id = sfcreate(sd_id, name, number_type, rank, dim_sizes)
status = sfwdata(sds_id, start, stride, edge, data)
```

A ***slab*** is an n-dimensional array whose dimensions are smaller than or equal to those of the SDS array into which it will be written. A slab is defined by the n-dimensional coordinate of its initial vertex and the lengths of each dimension.

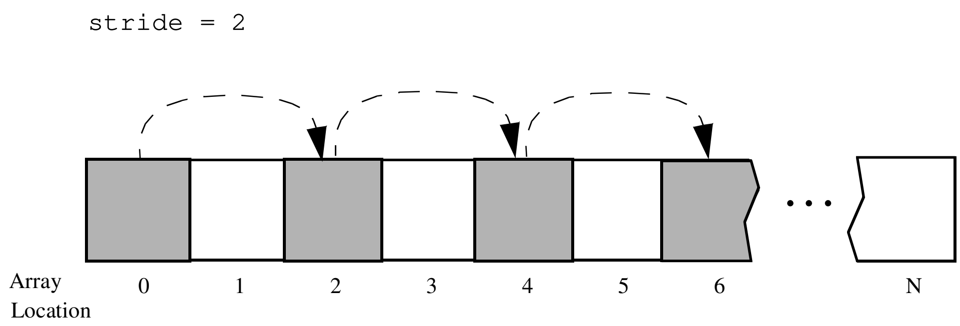
SDwritedata takes five arguments: `sds_id`, `start`, `stride`, `edge`, and `data`. The `sds_id` argument is the SDS identifier returned by **SDcreate** or **SDselect**. The arguments `start`, `stride`, and `edge` respectively describe the n-dimensional coordinates the SD interface will begin the write operation in the data set, the number of locations the current SDS location will be moved forward after each write, and the length of each dimension of the n-dimensional slab to be written. If the SDS array is smaller than the `data` argument array, the amount of data written will be limited to the maximum size of the SDS array.

The argument `start` is an array specifying the location in the data set array the write operation will begin. The indices are relative to 0 so the first data value of an array would have index `{0,0, ... 0}`. The size of `start` must be the same as the number of dimensions in the SDS array. In addition, each value in `start` must be smaller than its corresponding SDS array dimension unless the dimension is unlimited. Violating any of these conditions causes a termination of the write operation and causes an error condition to be generated.

The argument `stride` is an array specifying, for each dimension, the interval between values to be written. For example, setting `stride[0]=2` writes data to every other location along the first dimension.(See Figure 3b on page 30.) Setting `stride[0]=1` writes data to every location along the first dimension. If `stride` is defined as `NULL` in C or `0` in Fortran-77, it is assumed to contain all ones. For better performance, is it advised that you define the value of `stride` as `NULL` rather than setting it equal to 1. The length of the `stride` array must be the same as the number of dimensions in the SDS array. Also, each value in `stride` must be smaller than or equal to its corresponding SDS array dimension unless the dimension is unlimited. A violation of any of these conditions terminates the write operation and causes an error condition to be generated.

FIGURE 3b

Description of "Strides"



The argument `edge` is an array specifying the length of each dimension of the slab to be written. If the rank of the slab is less than that of the SDS data set, all additional dimensions must be specified as 1. Each value in the `edge` array must not be larger than the length of the corresponding dimension in the SDS data set. Attempting to read or write slabs larger than the size of the SDS data set will result in an error condition. The size of `edge` must be equal to the number of dimensions in the SDS array. In addition, the sum of each value in the `edge` array and the corresponding value in the `start` array must be smaller than or equal to its corresponding SDS array dimension unless the dimension is unlimited. A violation of any of these conditions terminates the write operation and results in an error condition.

Be aware that the mapping between the dimensions of a slab and the order in which the slab values are stored in memory is different between C and Fortran-77. In C the values are stored with the assumption that the last dimension of the slab varies fastest (or "row-major order" storage), but in Fortran-77 the first dimension varies fastest (or "column-major order" storage). These storage order conventions can cause some confusion when data written by a C program is read by a Fortran-77 program or vice versa.

There are two Fortran-77 versions of this routine: **sfwdata** and **sfwcdata**. The **sfwdata** routine writes numeric scientific data and **sfwcdata** writes character scientific data.

The parameters of **SDwritedata** are described in the following table. Note that, because there are two Fortran-77 versions of **SDwritedata**, there are correspondingly two entries in the “Data Type” field of the *data* parameter.

TABLE 3D

SDwritedata Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDwritedata (sfwdata/ sfwcdata)	sds_id	int32	integer	Data set identifier.
	start	int32 []	integer (*)	Array containing the position the write will start for each dimension.
	stride	int32 []	integer (*)	Array containing the number of data locations the current location is to be moved forward before the next write.
	edge	int32 []	integer (*)	Array containing the number of data elements that will be written along each dimension.
	data	VOIDP	<valid numeric data type>	Buffer for the data to be written.

3.5.2.1 Filling an Entire Array

Filling an array is a simple slab operation where the slab begins at the origin of the SDS array and fills every location in the array. **SDwritedata** will fill an SDS array with data when given the origin (`start={0,0, ... 0}`) as its starting coordinates, a stride value of `NULL`, and edge dimensions equal to the size of the SDS array (`edge={dim_sizes[0], dim_sizes[1], ... dim_size[rank-1]}`).

EXAMPLE 3. **Creating and Writing to an SDS**

These examples use **SDcreate** under the C interface and **sfcreate** under the Fortran-77 interface. The only difference between writing array data to an existing SDS and writing it to a newly created array is the use of **SDselect** instead of **SDcreate**.

C:

```
#include "hdf.h"
#include "mfhdf.h"

#define X_LENGTH 5
#define Y_LENGTH 16

main( )
{
int32 sd_id, sds_id, status;
int32 dims[2], start[2], edges[2], rank;
int16 array_data[Y_LENGTH][X_LENGTH];
intn i, j;

/* Create and open the file and initiate the SD interface. */
sd_id = SDstart("Example3.hdf", DFACC_CREATE);

/* Define the rank and dimensions of the data set to be created. */
rank = 2;
dims[0] = Y_LENGTH;
dims[1] = X_LENGTH;

/* Create the array data set. */
sds_id = SDcreate(sd_id, "Ex_array_3", DFNT_INT16, rank, dims);

/* Fill the stored-data array with values. */
for (j = 0; j < Y_LENGTH; j++) {
    for (i = 0; i < X_LENGTH; i++)
        array_data[j][i] = (i + j) + 1;
}

/* Define the location, pattern, and size of the data set */
for (i = 0; i < rank; i++) {
    start[i] = 0;
    edges[i] = dims[i];
}

/* Write the stored data to the "Ex_Array_3" data set. The fifth
* argument must be explicitly cast to a generic pointer to conform
* to the HDF API definition for SDwritedata. */
status = SDwritedata(sds_id, start, NULL, edges, (VOIDP)array_data);

/* Terminate access to the array. */
status = SDendaccess(sds_id);

/* Terminate access to the SD interface and close the file. */
```

```
status = SDend(sd_id);

}
```

FORTRAN:

```
     PROGRAM FILLED ARRAY

      integer*4 sd_id, sds_id, rank
      integer dims(2), start(2), edges(2), stride(2), status
      integer*2 i, j
      integer sfstart, sfcreate, sfwdata, sfendacc, sfend

C     DFACC_CREATE and DFNT_INT16 are defined in hdf.h.
      integer*4 DFACC_CREATE, DFNT_INT16
      integer*4 X_LENGTH, Y_LENGTH
      parameter (DFACC_CREATE = 4, DFNT_INT16 = 22, X_LENGTH = 5,
     +           Y_LENGTH = 16)
      integer*2 array_data(X_LENGTH, Y_LENGTH)

C     Create and open the file and initiate the SD interface.
      sd_id = sfstart('Example3.hdf', DFACC_CREATE)

C     Define the rank and dimensions of the data set to be created.
      rank = 2
      dims(1) = X_LENGTH
      dims(2) = Y_LENGTH

C     Create the data set.
      sds_id = sfcreate(sd_id, 'Ex_array_3', DFNT_INT16, rank, dims)

C     Fill the stored-data array with values.
      do 20 j = 1, Y_LENGTH
         do 10 i = 1, X_LENGTH
            array_data(i, j) = i + j - 1
10         continue
20    continue

C     Define the location, pattern, and size of the data set
C     that will be written to.
      start(1) = 0
      start(2) = 0
      edges(1) = X_LENGTH
      edges(2) = Y_LENGTH
      stride(1) = 1
      stride(2) = 1

C     Write the stored data to the "Ex_array_3" data set.
      status = sfwdata(sds_id, start, stride, edges, array_data)

C     Terminate access to the array.
      status = sfendacc(sds_id)

C     Terminate access to the SD interface and close the file.
      status = sfend(sd_id)

      end
```

3.5.2.2 Writing Slabs to an SDS Array

To allow preexisting data to be modified, the HDF library does not prevent **SDwritedata** from overwriting one slab with another. As a result, the calling program is responsible for managing any overlap when writing slabs. The HDF library will issue an error if a slab extends past the valid boundaries of the SDS data set, although appending data along an unlimited dimension is allowed.

EXAMPLE 4.

Writing a Slab of Data to an SDS

These examples show how to write a series of slabs to a data set. The programs create a three-dimensional array with the size of the x-dimension being four elements, the y-dimension five elements and the z-dimension six elements. As the data set elements are written, the slabs are "sliced" along the y-dimension.

Note that the *start*, *edge* and *stride* arguments in **SDwritedata** refer to the coordinate locations in the file representation of the data set, not the location within the `write_data` array. Therefore, the actual slab to be written is first buffered in the `zx_data` array.

In most real-world HDF-based applications that perform slab writes, the `write_data` array would be a buffer area for data previously read in by **SDreaddata**. In this example, this data read step has been omitted so that this example can focus on the slab write procedure.

C:

```
#include "hdf.h"
#include "mfhdf.h"

#define X_LENGTH 4
#define Y_LENGTH 5
#define Z_LENGTH 6

main( )
{
int32 sd_id, sds_id, rank, status;
int32 dims[3], start[3], edges[3];
int16 write_data[Z_LENGTH][Y_LENGTH][X_LENGTH];
int16 zx_data[Z_LENGTH][X_LENGTH];
intn i, j, k;

/* Open the file. */
sd_id = SDstart("Example4.hdf", DFACC_CREATE);

/* Define the rank and dimensions of the array to be created. */
rank = 3;
dims[0] = Z_LENGTH;
dims[1] = Y_LENGTH;
dims[2] = X_LENGTH;

/* Create the array. */
sds_id = SDcreate(sd_id, "Ex_array_4", DFNT_INT16, rank, dims);

/* Compute and store the values that will be written to the data
   set. Fill the write_data array along the x-axis first. */
for (k = 0; k < Z_LENGTH; k++)
    for (j = 0; j < Y_LENGTH; j++)
        for (i = 0; i < X_LENGTH; i++)
            write_data[k][j][i] = (i + 1) + (j + 1) + (k + 1);

/* Within each for loop, set the start and edge parameters to write \
a 4-by-6 element slab of stored data from the write_data array to \
the data set. */
edges[0] = Z_LENGTH;
edges[1] = 1;
edges[2] = X_LENGTH;
start[0] = start[2] = 0;
```

```
for (j = Y_LENGTH; j > 0; j--) {
    start[1] = j - 1;
    for ( k = 0; k < Z_LENGTH; k++) {
        for ( i = 0; i < X_LENGTH; i++) {
            zx_data[k][i] = write_data[k][j-1][i];
            status = SDwritedata(sds_id, start, NULL, edges,
                         (VOIDP)zx_data);
                }
            }
}

/* Terminate access to the data set. */
status = SDendaccess(sds_id);

/* Terminate access to the SD interface and close the file. */
status = SDend(sd_id);

}
```

FORTRAN:

```
     PROGRAM WRITE SLAB

      integer*4 sd_id, sds_id, rank
      integer dims(3), start(3), edges(3), stride(3)
      integer i, j, k, status
      integer sfstart, sfcreate, sfwdata, sfendacc, sfend

C     DFACC_CREATE and DFNT_INT16 are defined in hdf.h.
      integer*4 DFACC_CREATE, DFNT_INT16
      integer*4 X_LENGTH, Y_LENGTH, Z_LENGTH
      parameter (DFACC_CREATE = 4, DFNT_INT16 = 22, X_LENGTH = 4,
     +           Y_LENGTH = 5, Z_LENGTH = 6)

      integer*2 write_data(X_LENGTH, Y_LENGTH, Z_LENGTH)
      integer*2 zx_data(X_LENGTH, Z_LENGTH)

C     Create the file.
      sd_id = sfstart('Example4.hdf', DFACC_CREATE)

C     Define the rank and dimensions of the array to be created.
      rank = 3
      dims(1) = X_LENGTH
      dims(2) = Y_LENGTH
      dims(3) = Z_LENGTH

C     Create the array.
      sds_id = sfcreate(sd_id, 'Ex_array_4', DFNT_INT16, rank, dims)

C     Compute and store the values that will be later written to the
C     selected array data set. Fill the array_data array along the
C     x-axis first.
      do 30 k = 1, Z_LENGTH
         do 20 j = 1, Y_LENGTH
            do 10 i = 1, X_LENGTH
               write_data(i, j, k) = i + j + k
10           continue
20        continue
30    continue

C     Within each do loop, set the start and edge parameters so that
C     a 4-by-6 element slab of stored data will be written from the
```

```
C     array_data array to the data set.
      edges(1) = X_LENGTH
      edges(2) = 1
      edges(3) = Z_LENGTH
      start(1) = 0
      start(3) = 0
      stride(1) = 1
      stride(2) = 1
      stride(3) = 1

      do 60 j = Y_LENGTH, 0, -1
       start(2) = j - 1
       do 50 k = 1, Z_LENGTH
        do 40 i = 1, X_LENGTH
         zx_data(i, k) = write_data(i, j, k)
         status = sfwdata(sds_id, start, stride, edges, zx_data)
40      continue
50     continue
60    continue

C     Terminate access to the data set.
      status = sfendacc(sds_id)

C     Terminate access to the SD interface and close the file.
      status = sfend(sd_id)

      end
```

EXAMPLE 5.

Altering Values Within an Array Data Set

These examples demonstrate the procedure for changing the value of one element, located in the second column and tenth row, of a data set.

This procedure can be used to alter the values of a group of elements within the data set by changing the `start` and `edge` arguments of the second call of the **SDwritedata** function. Notice that a `NULL` value is passed into **SDwritedata** in the C program instead of the `stride` array.

C:

```
#include "hdf.h"
#include "mfhdf.h"

#define X_LENGTH 5
#define Y_LENGTH 16

main( )
{
int32 sd_id, sds_id, status;
int32 start[2], edges[2];
int16 write_data[Y_LENGTH][X_LENGTH], intval;
intn i, j;

/*  Open the file. */
sd_id = SDstart("Example3.hdf", DFACC_RDWR);

/*  Select the first data set. */
sds_id = SDselect(sd_id, 0);

/* Compute and store the values that will be written to the
   selected array data set. */
for (j = 0; j < Y_LENGTH; j++)
```

```
    for (i = 0; i < X_LENGTH; i++)
        write_data[j][i] = (i + 1) + (j + 1) * 10;

/* Set up the start and edge parameters to write the buffered
   data to the entire array data set. */
start[0] = start[1] = 0;
edges[0] = Y_LENGTH;
edges[1] = X_LENGTH;

/* Write the buffered data to the "Ex_array_3" data set. The fifth
   argument must be explicitly cast to a generic pointer to conform
   to the HDF API definition of SDwritedata.*/
status = SDwritedata(sds_id, start, NULL, edges, (VOIDP)write_data);

/* Alter a value within this data set. */
intval = 15;
start[0] = 10;
start[1] = 1;
edges[0] = 1;
edges[1] = 1;
status = SDwritedata(sds_id, start, NULL, edges, (VOIDP)&intval);

/* Terminate access to the data set. */
status = SDendaccess(sds_id);

/* Terminate access to the SD interface and close the file. */
status = SDend(sd_id);

}
```

FORTRAN:

```
     PROGRAM ALTER DATA

      integer*4 sd_id, sds_id
      integer start(2), edges(2), stride(2)
      integer i, j, status
      integer sfstart, sfselect, sfwdata, sfendacc, sfend

C     DFACC_RDWR is defined in hdf.h.
      integer*4 DFACC_RDWR
      integer*4 X_LENGTH, Y_LENGTH
      parameter (DFACC_RDWR = 3, X_LENGTH = 5, Y_LENGTH = 16)

      integer*2 array_data(X_LENGTH, Y_LENGTH), intval

C     Open the file and initiate the SD interface.
      sd_id = sfstart('Example3.hdf', DFACC_RDWR)

C     Select the first data set.
      sds_id = sfselect(sd_id, 0)

C     Compute and store the values that will be later written to
C     the data set.
      do 20 j = 1, Y_LENGTH
         do 10 i = 1, X_LENGTH
            array_data(i, j) = i + j * 10
10         continue
20    continue

C     Initialize the start, edge and stride parameters to write the
C     stored data to the data set.
      start(1) = 0
```

```
      start(2) = 0
      edges(1) = X_LENGTH
      edges(2) = Y_LENGTH
      stride(1) = 1
      stride(2) = 1

C     Write the stored data to the data set.
      status = sfwdata(sds_id, start, stride, edges, array_data)

C     Alter the value of the data set element in the first column,
C     tenth row to be 15.
      intval = 15
      start(1) = 1
      start(2) = 10
      edges(1) = 1
      edges(2) = 1
      status = sfwdata(sds_id, start, stride, edges, intval)

C     Terminate access to the data set.
      status = sfendacc(sds_id)

C     Terminate access to the SD interface and close the file.
      status = sfend(sd_id)

      end
```

3.5.2.3 Appending Data to an SDS Array Using the Unlimited Dimension

If one dimension of an SDS array must be extendable, it is possible to create an ***appendable SDS array***. An SDS array is appendable if one dimension is specified as an "unlimited" when the array is created.

In C, if the **SDcreate** parameter `dim_sizes[0]` is assigned the value `SD_UNLIMITED` it is considered to be an unlimited dimension. This is the only dimension that may be specified as unlimited in a C program and can also be understood as the *first* dimension or the dimension of the *lowest* rank value, of the SDS. In Fortran-77, only the *last* dimension or the dimension of the *highest* rank value, can be unlimited. In other words, in Fortran-77 `dim_sizes(rank)` must be set to the value `SD_UNLIMITED` to make the array appendable.

To append data to an extensible data set without overwriting data, specify the appropriate `start` coordinates in the **SDwritedata** routine. For example, if the current coordinate boundary of an unlimited dimension is defined as 15, appending data to the array without overwriting existing data requires a `start` coordinate of 16. To append data by overwriting data, specify a `start` coordinate less than the current boundary of the unlimited dimension. In either case, all but the unlimited coordinate in `start` must fall within the boundaries of the original array definition.

Any time a write operation is attempted beyond the current boundary, the HDF library will automatically adjust the dimension record to the new length. If the new data both begins and ends past the boundary of the array, locations between the existing boundary and the beginning of the new data stream are initialized to the assigned fill value if there is one or the default fill value if none is assigned.

EXAMPLE 6. **Appending Data to an SDS Array Using an Unlimited Dimension**

In the C example, the length of the SDS array's y-dimension (or the first dimension) is set to the value `SD_UNLIMITED`, which defines the x-dimension (or the second and last dimension) as appendable. In the Fortran-77 version, the length of the y-dimension (or the second and last dimension) is set to the value SD_UNLIMITED. Again, this is because only the dimension of the highest rank (in this case the x-dimension is of rank 1 and the y-dimension is of rank 2) can be defined as appendable under the Fortran-77 interface.

C:

```
#include "hdf.h"
#include "mfhdf.h"

#define X_LENGTH 10
#define Y_LENGTH 10

main( )
{

int32 sd_id, sds_id, sds_idx;
int32 dims[2], rank;
int16 array_data[Y_LENGTH][X_LENGTH], append_data[X_LENGTH];
int32 start[2], edges[2];
intn i = 0, j, status;

/* Open the file and initiate the SD interface. */
sd_id = SDstart("Example3.hdf", DFACC_RDWR);

/* Define the rank and dimensions of the array. Make the first \
   array dimension appendable by defining it's length to be \
   unlimited.*/
rank = 2;
dims[0] = SD_UNLIMITED;
dims[1] = X_LENGTH;

/* Create the array data set. */
sds_id = SDcreate(sd_id, "Ex_File_6", DFNT_INT16, rank, dims);

/* Store the array values. */
for (j = 0; j < Y_LENGTH; j++) {
    for (i = 0; i < X_LENGTH; i++)
        array_data[j][i] = (i + 1) + (j + 1);
}

/* Write the data to the array. */
start[0] = start[1] = 0;
edges[0] = Y_LENGTH;
edges[1] = X_LENGTH;

/* Perform the initial write to the array data set. */
status = SDwritedata(sds_id, start, NULL, edges, (VOIDP)array_data);

/* Terminate access to the array data set, terminate access \
to the SD interface and close the file. */
status = SDendaccess(sds_id);
status = SDend(sd_id);

/* Reopen the file and initiate the SD interface in preparation
for appending data to the data set. Then select the first
data set.*/
sd_id = SDstart("Example3.hdf", DFACC_RDWR);
sds_idx = SDnametoindex(sd_id, "Ex_File_6");
sds_id = SDselect(sd_id, sds_idx);

/* Store the array values to be appended to the data set. */
for (i = 0; i < X_LENGTH; i++)
    append_data[i] = i + 1;
```

```
/* Define the location of the append to start at the first column \
of the sixteenth row of the data set and to stop at the end of the \
fifth row. */
start[0] = Y_LENGTH;
start[1] = 0;
edges[0] = 1;
edges[1] = X_LENGTH;

/* Append the stored data to the array data set. */
status = SDwritedata(sds_id, start, NULL, edges, (VOIDP)append_data);

/* Terminate access to the array data set. */
status = SDendaccess(sds_id);

/* Terminate access to the SD interface and close the file. */
status = SDend(sd_id);

}
```

FORTRAN:

```
     PROGRAM APPEND DATA

      integer*4 sd_id, sds_id, sds_idx
      integer dims(2), rank
      integer start(2), edges(2), stride(2)
      integer status
      integer sfstart, sfcreate, sfwdata, sfn2index, sfselect
      integer sfendacc, sfend

C     DFACC_RDWR and DFNT_INT16 are defined in hdf.h, SD_UNLIMITED
C     is defined in hdf.h.
      integer*4 DFACC_RDWR, DFNT_INT16, SD_UNLIMITED
      integer*4 X_LENGTH, Y_LENGTH
      parameter (DFACC_RDWR = 3, DFNT_INT16 = 22, SD_UNLIMITED = 0,
     +           X_LENGTH = 10, Y_LENGTH = 10)

      integer*2 array_data(X_LENGTH, Y_LENGTH)
      integer*2 append_data(X_LENGTH)
      integer*2 i, j

C     Open the file and initiate the SD interface.
      sd_id = sfstart('Example3.hdf', DFACC_RDWR)

C     Define the rank and dimensions of the array. Make the
C     last dimension appendable by defining it's length as unlimited.
      rank = 2
      dims(1) = X_LENGTH
      dims(2) = SD_UNLIMITED

C     Create the array data set.
      sds_id = sfcreate(sd_id, 'Ex_File_6', DFNT_INT16, rank, dims)

C     Store the array values.
      do 20 j = 1, Y_LENGTH
         do 10 i = 1, X_LENGTH
            array_data(i, j) = i + j
10       continue
20    continue

C     Write the data to the array.
      start(1) = 0
```

```
      start(2) = 0
      edges(1) = X_LENGTH
      edges(2) = Y_LENGTH
      stride(1) = 1
      stride(2) = 1

C     Perform the initial write to the data set.
      status = sfwdata(sds_id, start, stride, edges, array_data)

C     Terminate access to the data set, terminate access
C     to the SD interface and close the file.
      status = sfendacc(sds_id)
      status = sfend(sd_id)

C     Reopen the file and initiate the SD interface in preparation
C     for appending data. Then select the first data set.
      sd_id = sfstart('Example3.hdf', DFACC_RDWR)
      sds_idx = sfn2index(sd_id, 'Ex_File_6')
      sds_id = sfselect(sd_id, sds_idx)

C     Store the array values to be appended to the data set.
      do 30 i = 1, X_LENGTH
         append_data(i) = i
30    continue

C     Define the location of the append to start at the first
C     column of the third row and to stop at the end of the third row.
      start(1) = 0
      start(2) = Y_LENGTH
      edges(1) = X_LENGTH
      edges(2) = 1

C     Append the stored data to the data set.
      status = sfwdata(sds_id, start, stride, edges, append_data)

C     Terminate access to the array data set.
      status = sfendacc(sds_id)

C     Terminate access to the SD interface and close the file.
      status = sfend(sd_id)

      end
```

3.5.3 Compressing SD SDS Array Data: SDsetcompress

Uncompressed SDS array data is compressed, or new compressed data sets are created, by calling the **SDsetcompress** routine. **SDsetcompress** compresses the data set data at the time it is called (not during the next call to **SDwritedata**)and supports all standard HDF compression algorithms. These algorithms are:

- JPEG.
- Adaptive Huffman.
- GZIP "deflation". (Lempel/Ziv-77 dictionary coder)
- Run-length encoding.

In the future, the following algorithms may be included: Lempel/Ziv-78 dictionary coding, an arithmetic coder, and a faster Huffman algorithm.

The **SDsetcompress** routine 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. All essential compression functionality is provided by **SDsetcompress**.

The syntax of the **SDsetcompress** routine is as follows:

C:
```
status = SDsetcompress(sds_id, comp_type, c_info);
```

The `comp_type` parameter is the compression type definition and is set to `COMP_CODE_JPEG` for JPEG compression, `COMP_CODE_RLE` for run-length encoding, `COMP_CODE_DEFLATE` for Gnu ZIP (or GZIP) compression, `COMP_CODE_SKPHUFF` for skipping Huffman or `COMP_CODE_NONE` for no compression.

The `c_info` parameter is a pointer to a union structure of type `coder_info`. If `comp_type` is set to `COMP_CODE_NONE` or `COMP_CODE_RLE`, this is unused and can be set to `NULL`. If it's set to `COMP_CODE_SKPHUFF`, the `skphuff` structure in the `coder_info` union must be provided information about the size, in bytes, of the data elements. If it's set to `COMP_CODE_DEFLATE`, the deflate structure in the `coder_info` union must be provided information about the compression "effort".

For example, to compress unsigned 16-bit integer data using the adaptive Huffman algorithm, the following definition and **SDsetcompress** call is used.

```
coder_info c_info;
c_info.skphuff.skp_size = sizeof(uint16);
status = SDsetcompress(sds_id, COMP_CODE_SKPHUFF, &c_info);
```

To compress a data set using the gzip deflation algorithm, with the maximum "effort" level designated, the following definition and **SDsetcompress** call is used.

```
coder_info c_info;
c_info.deflate_level = 9;
status = SDsetcompress(sds_id, COMP_CODE_DEFLATE, &c_info);
```

To compress a data set using the JPEG algorithm, with the quality level of 80 designated and the "force baseline" parameter enabled, the following definition and **SDsetcompress** call is used.

```
coder_info c_info;
c_info.quality = 80;
c_info.force_baseline = TRUE;
status = SDsetcompress(sds_id, COMP_CODE_JPEG, &c_info);
```

SDsetcompress functionality is currently limited to creating new datasets or appending new slabs onto existing datasets. Overwriting existing dataset data will be supported in the future.

Note that there is currently no Fortran-77 version of the **SDsetcompress** routine. It will be included in a future release.

TABLE 3E

SDsetcompress Parameter List

Routine Name	Parameter	Data Type	Description
		C	

SDsetcompress	sds_id	int32	Data set identifier.
	comp_type	char *	Compression method.
	cinfo	comp_info *	Pointer to compression information structure.

3.5.3.1 Rules for Writing to a Compressed Data Set

Due to certain limitations in the way that compressed data sets are stored, they aren't writable in the way that uncompressed data sets are. The "rules" for writing to a compressed data set are the following:

1. Write the compressed data, in its entirety, to the data set. Build the data set in-core, then write it to the data set in a single write operation.
2. Append to a compressed data set. In other words, write to a compressed data set along it's unlimited dimension. If an unlimited dimension hasn't been defined for the data set, it can't be appended to in this way.
3. Write the compressed data, in its entirety, to any chunk in a chunked SDS.

These rules imply that it is impossible to overwrite subsets of non-chunked data sets. This is because the existing compression algorithms supported by HDF don't allow partial modification to a compressed datastream.

3.5.4 External File Operations

An ***external SDS array*** is an array stored in a file separate from the file containing the metadata for the array. With external arrays, it is possible to link data sets in the same HDF file to multiple external files or data sets in different HDF files to the same external file. Routines for writing external SDS arrays are only available in the C interface and can only be used with HDF files. Unidata-formatted netCDF files are not supported by these routines.

External arrays are functionally identical to arrays in the primary data file. During slab operations, the HDF library keeps track of the beginning of the data set and adds slabs at the appropriate position in the external file. When data is written or appended along a specified dimension, the HDF library writes along that dimension in the external file and updates the appropriate dimension record in the primary file.

There are two methods for creating external SDS arrays. The user can create a new data set in an external file or move data from an existing internal data set to an external file. In either case, only the array values are stored externally, all other data set information remains in the primary HDF file. When an external array is created, a sufficient amount of space is reserved in the external file for the entire data set. The data set will begin at the specified byte offset and extend the length of the data set. The write operation will overwrite the target locations in the external file. The external file may be of any format, provided the data types, byte ordering, and dimension ordering are supported by HDF. The primary file must be an HDF file.

3.5.4.1 Specifying the Directory Search Path of an External File: HXsetdir

There are three filesystem locations the HDF external file routines check when determining the location of an external file. They are, in order of precedence:

1. The directory path specified by the last call to the **HXsetdir** routine.
2. The directory path specified by the $HDFEXTDIR shell environment variable.
3. The locations searched by the standard **open(3)** routine.

Until the **HXsetdir** routine is called, either the directories specified by the $HDFEXTDIR environment variable are searched if the variable has been set, or Item 3 will be the search method used. If the **HXsetdir** routine hasn't been called and the $HDFEXTDIR variable hasn't been set, Item 3 will again be the search method used. Setting the $HDFEXTDIR environment variable effectively ensures that Item 3 will never be used as a default method as it can't be unset within a calling program.

HXsetdir has one argument, a string specifying the directory list to be searched. This list can consist of one directory name or a set of directory names separated by colons. If **HXsetdir** is passed a null string and the $HDFEXTDIR environment variable has been set, the directories specified by $HDFEXTDIR are searched, or Item 3 will be the search method used if $HDFEXTDIR hasn't been set.

If an error condition is encountered, **HXsetdir** leaves the directory search path unchanged. The directory search path specified by **HXsetdir** remains in effect throughout the scope of the calling program.

The parameters of **HXsetdir** are described further in the following table.

TABLE 3F **HXsetdir Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
HXsetdir (**hxsdir**)	dir_list	char *	character* (*)	Directory list to be searched.
	dir_length	None	integer	Length of the `dir_list` string.

3.5.4.2 Specifying the Location of the Next External File to be Created: HXsetcreatedir

HXsetcreatedir specifies the directory location of the next external file to be created. It overrides the directory location specified by the $HDFEXTCREATEDIR and the locations searched by the **open(3)** call in the same manner as **HXsetdir**. Specifically, the precedence is:

1. The directory specified by the last call to the **HXsetcreatedir** routine.
2. The directory specified by the $HDFEXTCREATEDIR shell environment variable.
3. The locations searched by the standard **open(3)** routine.

HXsetcreatedir has one argument: the directory location of the next external file to be created. If an error is encountered, the directory location is left unchanged.

The parameters of **HXsetcreatedir** are described further in the following table.

TABLE 3G **HXsetcreatedir Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
HXsetcreatedir (**hxsdir**)	dir	char *	character* (*)	Directory location of the next external file to be created.
	dir_length	None	integer	Length of the `dir` string.

3.5.4.3 Creating a Data Set in an External File: SDsetexternalfile

Creating a data set using external data involves the following steps:

1. Create the array.
2. Specify that an external data file is to be used.
3. Write data to the array.
4. Terminate access to the data set.

To create a data set containing an external file, the calling program must make the following calls.

C:
```
sds_id = SDcreate(sd_id, name, number_type, rank, dim_sizes);
status = SDsetexternalfile(sds_id, filename, offset);
status = SDwritedata(sds_id, start, stride, edge, data);
status = SDendaccess(sds_id);
```

FORTRAN:
```
sds_id = sfcreate(sd_id, name, number_type, rank, dim_sizes)
status = sfsextf(sds_id, filename, offset)
status = sfwdata(sds_id, start, stride, edge, data)
status = sfendacc(sds_id)
```

SDsetexternalfile marks the SDS identified by `sds_id` as one whose data is to be written to an external file. The parameter `filename` is the name of the external data file, and `offset` is the number of bytes from the beginning of the external file to the location where the first byte of data should be written.

When used in conjunction with **SDcreate**, **SDsetexternalfile** does not actually write data to an external file. Instead it marks the data set as an external data set for all subsequent **SDwritedata** operations. **SDsetexternalfile** can only be called once after a data set has been created.

If a file with the same name as `filename` exists in the current directory search path, HDF will access it as the external file. If the file does not exist, HDF will create one in the directory named in the last call to **HXsetcreatefile**. If an absolute pathname is specified, the external file will be created at the location specified by the pathname, overriding the location specified by the last call to **HXsetcreatefile**.

Once the name of an external file is established, it is impossible to change it without breaking the association between the data set's metadata and the data it describes.

Use caution when writing to existing external or primary files as the HDF library starts the write operation at the specified offset without checking if data is being overwritten.

For more information on the parameters used in **SDsetexternalfile** refer to the following table.

TABLE 3H **SDsetexternalfile Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDsetexternalfile (sfsextf)	sds_id	int32	integer	Data set identifier.
	filename	char *	character* (*)	Name of the external data set.
	offset	int32	integer	Offset in bytes from the beginning of the external file to the SDS data.

3.5.4.4 Moving Data to an External File

Data can be moved from a primary file to an external file. To do so requires the following steps:

1. Select the array.
2. Specify the external data file.

3. Terminate access to the data set.

To move data to an external file, the calling program must make the following calls:

C:

```
sds_id = SDselect(sd_id, sds_index);
status = SDsetexternalfile(sds_id, filename, offset);
status = SDendaccess(sds_id);
```

FORTRAN:

```
sds_id = sfcreate(sd_id, name, number_type, rank, dim_sizes)
status = sfsextf(sds_id, filename, offset)
status = sfendacc(sds_id)
```

When **SDsetexternalfile** is used in conjunction with **SDselect**, it will immediately write the existing data to the external file. Any data in the external file that occupies the space reserved for the external array will be overwritten as a result of this operation. Data can only be moved to an external file once after the parent data set has been created and the data set must first exist in the primary HDF file. During the operation, the data is written to the external file as a contiguous stream regardless of how it is stored in the primary file. Because data is moved "as is", any unwritten locations in the data set are preserved in the external file. Subsequent read and write operations performed on the data set will access the external file.

EXAMPLE 7.

Writing SDS Data to an HDF File Starting at an Offset

A subset of values stored in an SDS can be moved from an HDF file to an external file, as shown in the following examples.

C:

```
#include "hdf.h"

main( )
{

int32 sd_id, sds_id, offset, status;

/* Open the file. */
sd_id = SDstart("Example3.hdf", DFACC_RDWR);

/* Get the identifier for the first data set. */
sds_id = SDselect(sd_id, 0);

/* Create a file named "subfile1" and move the data set values into
 *  it, starting at byte location 24. */
offset = 24;
status = SDsetexternalfile(sds_id, "subfile1", offset);

/* Terminate access to the data set, SD interface and file. */
status = SDendaccess(sds_id);
status = SDend(sd_id);

}
```

FORTRAN:

```
      PROGRAM WRITE EXTFILE

       integer*4 sd_id, sds_id, offset, DFACC_RDWR
       integer status
       integer sfstart, sfselect, sfsextf, sfendacc, sfend
```

```
C     DFACC_RDWR is defined in hdf.h.
      parameter (DFACC_RDWR = 3)

C     Open the HDF file.
      sd_id = sfstart('Example3.hdf', DFACC_RDWR)

C     Get the identifier of the first data set.
      sds_id = sfselect(sd_id, 0)

C     Create a file named "subfile1" and move the data set
C     into it, starting at byte location 24.
      offset = 24
      status = sfsextf(sds_id, 'subfile1', offset)

C     Dispose of the data set identifier to terminate access.
      status = sfendacc(sds_id)

C     Dispose of the file identifier to close the file.
      status = sfend(sd_id)

      end
```

3.6 Reading Data from an SDS Array: SDreaddata

Selecting an SD data set and reading one or more slabs from it involves the following steps:

1. Select an SDS.
2. Read a slab or series of slabs.

To read data from an SDS array, the calling program must contain the following function calls:

C:
```
sds_id = SDselect(sd_id, sds_index);
status = SDreaddata(sds_id, start, stride, edge, data);
```

FORTRAN:
```
sds_id = sfselect(sd_id, sds_index)
status = sfrdata(sds_id, start, stride, edge, data)
```

SDreaddata reads an entire array, slabs of an array or it can be used to read a subset of the array. For more information on reading attributes or scales see Section 3.10.3 on page 84 and Section 3.9.4.3 on page 75.

The `sds_id` argument is the SDS id returned by **SDcreate** or **SDselect**. As with **SDwritedata,** the arguments `start`, `stride`, and `edge` respectively describe the n-dimensional coordinate of the slab the SD interface will begin the read operation, number of locations the current SDS location will be moved forward after each read, and the length of each dimension of the subset to be read. If the SDS array is smaller than the `data` argument array, the amount of data read will be truncated to the size of the SDS array. For additional information on the `start`, `stride` and `edge` parameters refer to Section 3.5.2 on page 28.

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.

The parameters of **SDreaddata** are further described in the following table. Note that, because there are two Fortran-77 versions of **SDreaddata**, there are correspondingly two entries in the “Data Type” field of the *data* parameter.

TABLE 3I

SDreaddata Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDreaddata (sfrdata/ sfrcdata)	sds_id	int32	integer	Data set identifier.
	start	int32 []	integer (*)	Array containing the position the read will start for each dimension.
	stride	int32 []	integer (*)	Array containing the number of data locations the current location is to be moved forward before the next read.
	edge	int32 []	integer (*)	Array containing the number of data elements that will be read along each dimension.
	data	VOIDP	<valid numeric data type>	Buffer the data will be read into.

EXAMPLE 8.

Reading an Entire SDS

When **SDreaddata** is used to read an entire n-dimensional SDS array, the coordinates for the start position must begin at 0 for each dimension (`start={0,0, ... 0}`), the interval between each read must equal 1 for each dimension (`stride=NULL or stride={1,1, ... 1}`), and the size of each dimension must equal the size of the array itself (`edge={dim_size_1, dim_size_2, ... dim_size_n}`). The `data` buffer must have enough preallocated space to hold the data.

C:

```
#include "hdf.h"
#include "mfhdf.h"

#define X_LENGTH 4
#define Y_LENGTH 5
#define Z_LENGTH 6

main( )
{
int32 sd_id, sds_id, status;
int32 start[3], edges[3];
int16 array_data[Z_LENGTH][Y_LENGTH][X_LENGTH];

/* Open the file and initiate the SD interface. */
sd_id = SDstart("Example4.hdf", DFACC_RDONLY);

/* Select the first (and in this case, only) data set in the file. */
sds_id = SDselect(sd_id, 0);

/* Define the location, pattern, and size of the data to read. */
start[0] = start[1] = start[2] = 0;
edges[0] = Z_LENGTH;
edges[1] = Y_LENGTH;
edges[2] = X_LENGTH;

/* Read the array. */
status = SDreaddata(sds_id, start, NULL, edges, (VOIDP)array_data);

/* Terminate access to the array */
status = SDendaccess(sds_id);

/* Terminate access to the SD interface and close the file. */
status = SDend(sd_id);

}
```

FORTRAN:

```
      PROGRAM SEARCH INDEX

       integer*4 gr_id, gr_index, ri_id, file_id, status
       integer*4 start(2), edges(2), stride(2)
       integer mgstart, mgn2ndx, mgrdimg, mgendac, mgselct
       integer mgend, hopen, hclose

C      DFACC_RDONLY is defined in hdf.h.
       integer*4 DFACC_RDONLY
       integer*4 X_LENGTH, Y_LENGTH
       parameter (DFACC_RDONLY = 1, X_LENGTH = 10, Y_LENGTH = 20)
       integer*2 image_data(2, X_LENGTH, Y_LENGTH)

C      Open the file.
       file_id = hopen('Example5.hdf', DFACC_RDONLY, 0)

C      Initialize the GR interface.
       gr_id = mgstart(file_id)

C      Search for the index of a non-existent image.
       gr_index = mgn2ndx(gr_id, 'Invalid_Data_Set_Name')

C      Error condition: gr_index contains the value -1.
C      Search for the index of the image named 'Image_array_5'.
       gr_index = mgn2ndx(gr_id, 'Image_array_5')

C      Select the image corresponding to the returned index.
       ri_id = mgselct(gr_id, gr_index)

C      Read the image data into the image_data array.
       start(1) = 0
       start(2) = 0
       edges(1) = X_LENGTH
       edges(2) = Y_LENGTH
       stride(1) = 1
       stride(2) = 1
       status = mgrdimg(ri_id, start, stride, edges, image_data)

C      Terminate access to the array.
       status = mgendac(ri_id)

C      Terminate access to the GR interface and the file.
       status = mgend(gr_id)

C      Close the file.
       status = hclose(file_id)

       end
```

EXAMPLE 9.

Reading a Subset of an SDS Array

The `start` and `edges` arguments of the **SDreaddata** function can be used to read a subset of the entire data set, as shown in the following examples.

C:

```
#include "hdf.h"
```

```
#include "mfhdf.h"

#define X_LENGTH 5
#define Y_LENGTH 16

main( )
{

int32 sd_id, sds_id;
int32 start[2], edges[2];
int16 all_data[Y_LENGTH][X_LENGTH], subset_data[7][3], status;

/* Open the file for read-only access. */
sd_id = SDstart("Example3.hdf", DFACC_RDONLY);

/* Select the first data set. */
sds_id = SDselect(sd_id, 0);

/* First, read the entire data set. */
start[0] = start[1] = 0;
edges[0] = Y_LENGTH;
edges[1] = X_LENGTH;
status = SDreaddata(sds_id, start, NULL, edges, (VOIDP)all_data);

/* Read a subset of the data set. */
start[0] = 1;
start[1] = 1;
edges[0] = 7;
edges[1] = 3;
status = SDreaddata(sds_id, start, NULL, edges, (VOIDP)subset_data);

/* Terminate access to the array. */
status = SDendaccess(sds_id);

/* Terminate access to the SD interface and close the file. */
status = SDend(sd_id);

}
```

FORTRAN:

```
     PROGRAM READ SUBSET

      integer*4 sd_id, sds_id
      integer start(2), edges(2), stride(2), status
      integer sfstart, sfselect, sfrdata, sfendacc, sfend

C     DFACC_RDONLY is defined in hdf.h.
      integer*4 DFACC_RDONLY
      integer*4 X_LENGTH, Y_LENGTH
      parameter (DFACC_RDONLY = 1, X_LENGTH = 5, Y_LENGTH = 16)
      integer*2 all_data(X_LENGTH, Y_LENGTH)
      integer*2 subset_data(3, 7)

C     Open the file.
      sd_id = sfstart('Example3.hdf', DFACC_RDONLY)

C     Select the first data set.
      sds_id = sfselect(sd_id, 0)

C     Read the entire data set.
      start(1) = 0
      start(2) = 0
```

```
      edges(1) = 5
      edges(2) = 16
      stride(1) = 1
      stride(2) = 1
      status = sfrdata(sds_id, start, stride, edges, all_data)

C     Read a subset from the middle of the data set.
      start(1) = 1
      start(2) = 1
      edges(1) = 3
      edges(2) = 7
      status = sfrdata(sds_id, start, stride, edges, subset_data)

C     Terminate access to the array.
      status = sfendacc(sds_id)

C     Terminate access to the SD interface and close the file.
      status = sfend(sd_id)

      end
```

EXAMPLE 10.

Sampling SDS Data

These examples demonstrate how samples of data set elements can be read by using the `stride` argument of the **SDreaddata** function. Here we read every tenth row and every other column.

C:

```
#include "hdf.h"
#include "mfhdf.h"

#define X_LENGTH 10
#define Y_LENGTH 20

main( )
{

int32 sd_id, sds_id, rank, start[2], edges[2], stride[2], dims[2];
int16 all_data[Y_LENGTH][X_LENGTH], sample_data[5][5];
intn i, j;
int16 status;

/* Open the file. */
sd_id = SDstart("Example10.hdf", DFACC_CREATE);

/* Define the rank and dimensions of the array to be created. */
rank = 2;
dims[0] = Y_LENGTH;
dims[1] = X_LENGTH;

/* Create the array. */
sds_id = SDcreate(sd_id, "Ex_array_10", DFNT_INT16, rank, dims);

/* Compute and store the data values. */
for (j = 0; j < Y_LENGTH; j++) {
    for (i = 0; i < X_LENGTH; i++)
        all_data[j][i] = i + j * 10;
}

/* Define the start and edge parameters. */
```

```
start[0] = start[1] = 0;
edges[0] = Y_LENGTH;
edges[1] = X_LENGTH;

/* Write the buffered data in all_data to the data set. */
status = SDwritedata(sds_id, start, NULL, edges, (VOIDP)all_data);

/* Close the SD interface and the file, then re-open both and \
select the first and only data set in the file. */

status = SDendaccess(sds_id);
status = SDend(sd_id);

sd_id = SDstart("Example10.hdf", DFACC_RDONLY);
sds_id = SDselect(sd_id, 0);

/* Read the data into the sample_data array, skipping every fourth \
row and every other column. */
start[0] = start[1] = 0;
edges[0] = 5;
edges[1] = 5;
stride[0] = 4;
stride[1] = 2;

status = SDreaddata(sds_id, start, stride, edges, (VOIDP)sample_data);

/* Terminate access to the array */
status = SDendaccess(sds_id);

/* Terminate access to the SD interface and close the file */
status = SDend(sd_id);

}
```

FORTRAN:

```
     PROGRAM READ STRIDES

      integer*4 sd_id, sds_id, rank
      integer*4 start(2), edges(2), stride(2), dims(2)
      integer i, j, status
      integer sfstart, sfcreate, sfwdata, sfrdata, sfendacc
      integer sfselect, sfend

C     DFACC_CREATE and DFNT_INT16 are defined in hdf.h.
      integer*4 DFACC_CREATE, DFACC_RDONLY, DFNT_INT16
      integer*4 X_LENGTH, Y_LENGTH
      parameter (DFACC_CREATE = 4, DFACC_RDONLY = 1, DFNT_INT16 = 22,
     +            X_LENGTH = 10, Y_LENGTH = 20)
      integer*2 all_data(X_LENGTH, Y_LENGTH), sample_data(5, 5)

C     Create the file.
      sd_id = sfstart('Example10.hdf', DFACC_CREATE)

C     Define the rank and dimensions of the array to be created.
      rank = 2
      dims(1) = X_LENGTH
      dims(2) = Y_LENGTH

C     Create the array.
      sds_id = sfcreate(sd_id, 'Ex_array_10', DFNT_INT16, rank, dims)

C     Compute and store the data values.
```

```
      do 20 j = 1, Y_LENGTH
         do 10 i = 1, X_LENGTH
            all_data(i, j) = i + j * 10
10          continue
20    continue

C     Set up the start and edge parameters to write the buffered
C     data to the entire array data set.
      start(1) = 0
      start(2) = 0
      edges(1) = X_LENGTH
      edges(2) = Y_LENGTH
      stride(1) = 1
      stride(2) = 1

C     Write the buffered data in wrt_data to the data set.
      status = sfwdata(sds_id, start, stride, edges, all_data)

C     Close the SD interface and the file, then re-open both and
C     select the first and only data set in the file.
      status = sfendacc(sds_id)
      status = sfend(sd_id)
      sd_id = sfstart('Example10.hdf', DFACC_RDONLY)
      sds_id = sfselect(sd_id, 0)

C     Read the data into the sample_data array, skipping every fourth
C     row and every other column.
      edges(1) = 5
      edges(2) = 5
      stride(1) = 2
      stride(2) = 4
      status = sfrdata(sds_id, start, stride, edges, sample_data)

C     Terminate access to the array.
      status = sfendacc(sds_id)

C     Terminate access to the SD interface and close the file.
      status = sfend(sd_id)

      end
```

3.6.1 Reading Data from an External File

SDS data is read from an external file in the same way that it is read from a primary file. Whether the SDS data is or is not stored in an external file is transparent to the user.

3.7 Obtaining Information About SD Data Sets

The routines covered in this section provide methods for obtaining information about the scientific data sets in a file, for identifying SDSs that meet certain criteria, and for obtaining information about the data sets themselves.

SDfileinfo obtains the number of SDSs and file attributes in a file, and **SDgetinfo** provides information about individual SDSs. To cycle through the data sets in a file, a calling program must use **SDfileinfo** to determine the number of data sets, followed by repeated calls to **SDgetinfo** to view them. The parameters of these two routines are described below. (See Table 3J on page 53.)

At times you might need to search through a file for an SDS for a given name or reference number. The **SDnametoindex** routine determines the index of an SDS from its name, and **SDreftoindex** determines the index of an SDS from its reference number. The parameters of these two routines are described below. (See Table 3K on page 56.)

3.7.1 Obtaining Information About the SDSs in a File: SDfileinfo

It is often useful to determine the number of scientific data sets and global SDS attributes contained in a file before executing a series of read or write operations. The **SDfileinfo** routine is designed for this purpose. To determine the contents of a file, the calling program must contain the following calls:

C:
```
sd_id = SDstart(filename, access_mode);
status = SDfileinfo(sd_id, n_datasets, n_file_attr);
status = SDend(sd_id);
```

FORTRAN:
```
sd_id = sfstart(filename, access_mode)
status = sffinfo(sd_id, n_datasets, n_file_attr)
status = sfend(sd_id)
```

SDfileinfo uses `n_datasets` to return the number of scientific data sets in the file, and `n_file_attr` to return the number of file attributes in the file.

3.7.2 Obtaining Information About a Specific SDS: SDgetinfo

Some information may be needed before reading and working with SDS arrays. For instance, if the rank, dimension sizes and/or data type of an array are unknown, it may be impossible to allocate the proper amount of memory to work with the array. The **SDgetinfo** routine provides basic information about SDS arrays.

SDgetinfo takes an SDS id as input, and returns the name, rank, dimensions, data type, and number of attributes for the corresponding SDS. The attribute count will only reflect the number of attributes assigned to the SDS array; file attributes are not included.

TABLE 3J **SDfileinfo and SDgetinfo Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDfileinfo (sffinfo)	file_id	int32	integer	File identifier.
	n_datasets	int32 *	integer	Number of data sets in the file.
	n_file_attr	int32 *	integer	Number of global attributes in the file.
SDgetinfo (sfginfo)	sds_id	int32	integer	Data set identifier
	sds_name	char*	character* (*)	Space to put the name of the data set.
	rank	int32 *	integer	Space to put the number of dimensions in the array.
	dim_sizes	int32 []	integer (*)	Space to put the size of each dimension in the array.
	data_type	int32 *	integer	Space to put the data type for the data in the array.
	nattrs	int32 *	integer	Space to put the number of attributes in the data set.

EXAMPLE 11. **Printing Data Set Names**

The **SDgetinfo** function can be called within a loop to retrieve the names of all data sets in an HDF file, as shown in the following examples.

C:

```
#include "hdf.h"
#include "mfhdf.h"

#include <stdio.h>

main( )
{

int32 sd_id, sds_id, n_datasets, n_file_attrs, index, status;
int32 dim_sizes[MAX_VAR_DIMS];
int32 rank, num_type, attributes;
char name[MAX_NC_NAME];

/* Open the file and initiate the SD interface. */
sd_id = SDstart("Example4.hdf", DFACC_RDONLY);

/* Determine the contents of the file. */
status = SDfileinfo(sd_id, &n_datasets, &n_file_attrs);

/* Access and print the name of every data set in the file. */
for (index = 0; index < n_datasets; index++) {
    sds_id = SDselect(sd_id, index);
    status = SDgetinfo(sds_id, name, &rank, dim_sizes, \
                &num_type, &attributes);

    printf("name = %s\n", name);
    status = SDendaccess(sds_id);
}

/* Terminate access to the SD interface and close the file. */
status = SDend(sd_id);

}
```

FORTRAN:

```
     PROGRAM PRINT NAMES

      integer*4 sd_id, sds_id
      integer*4 n_datasets, n_file_attrs, index
      integer status, attributes
      integer sfstart, sffinfo, sfselect, sfginfo
      integer sfendacc, sfend

C     DFACC_RDONLY is defined in hdf.h. MAX_NC_NAME and MAX_VAR_DIMS
C     are defined in netcdf.h.
      integer*4 DFACC_RDONLY, MAX_NC_NAME, MAX_VAR_DIMS
      parameter (DFACC_RDONLY = 1, MAX_NC_NAME = 256,
     +            MAX_VAR_DIMS = 32)

      integer*4 dim_sizes(MAX_VAR_DIMS)
      character name *(MAX_NC_NAME)

C     Open the file and initiate the SD interface.
      sd_id = sfstart('Example4.hdf', DFACC_RDONLY)

C     Determine the contents of the file.
      status = sffinfo(sd_id, n_datasets, n_file_attrs)

C     Access and print the names of every data set in the file.
```

```
      do 10 index = 0, n_datasets - 1
         sds_id = sfselect(sd_id, index)
         status = sfginfo(sds_id, name, rank, dim_sizes, num_type,
     +                    attributes)
         print *, "name = ", name
         status = sfendacc(sds_id)
10    continue

C     Terminate access to the SD interface and close the file.
      status = sfend(sd_id)

      end
```

3.7.3 Locating a SDS Data Set by Name: SDnametoindex

When an SDS is created, a unique number is assigned to it by the SD library so that it can be located in the file for future access. This number is referred to as the ***index*** of an SDS.

If the index of an SDS in an HDF file is known, **SDselect** can be used immediately. If not, it is necessary to determine the index by some other means. **SDnametoindex** determines the index of an SDS from its name. Selecting a data set by name involves the following steps:

1. Convert the SDS name into a valid index number.
2. Select the SDS by obtaining its identifier from its index number.

To access a SDS via its name, the calling program must contain the following function calls:

C:
```
sds_index = SDnametoindex(sd_id, sds_name);
sds_id = SDselect(sd_id, sds_index);
```

FORTRAN:
```
sds_index = sfn2index(sd_id, sds_name)
sds_id = sfselect(sd_id, sds_index)
```

SDnametoindex returns the index specified by the `sds_index` parameter for the first data set in the file with the name `sds_name`. If two data sets have the same name, **SDnametoindex** returns the index of the first data set. The index `sds_index` can then be used by **SDselect** to obtain an SDS id for the specified data set.

The **SDnametoindex** routine is case-sensitive and does not accept wildcards.

3.7.4 Locating an SDS by Reference Number: SDreftoindex

In addition to an index and name, data sets are also assigned a tag and reference number. **SDreftoindex** determines the index of the SDS from its reference number. Selecting a data set by reference number involves the following steps:

1. Convert the reference number for the SDS into a valid index number.
2. Select the SDS by obtaining its identifier from its index number.

To select a data set by reference number, the calling program must execute the following calls:

C:
```
sds_index = SDreftoindex(sd_id, ref);
sds_id = SDselect(sd_id, sds_index);
```

FORTRAN:
```
sds_index = sfref2index(sd_id, ref)
sds_id = sfselect(sd_id, sds_index)
```

SDreftoindex returns the index specified by the `sds_index` parameter for the SDS in the file with the reference number `ref`. The index `sds_index` can then be passed to **SDselect** to obtain an SDS id for the SDS.

Because the HDF library assigns reference numbers in unpredictable ways, reference numbers should be used to identify SDSs only when no other means are available. Reference numbers do not necessarily adhere to any ordering scheme.

TABLE 3K

SDnametoindex and SDreftoindex Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDnametoindex (sfn2index)	sd_id	int32	integer	SD interface identifier.
	sds_name	char *	character* (*)	Name of the data set to index.
SDreftoindex (sfref2index)	sd_id	int32	integer	SD interface identifier.
	ref	int32	integer	IN: Reference number for the specified data set.

EXAMPLE 12.

Searching for the Index of an SDS

In these examples, an invalid data set name is specified which results in **SDnametoindex** returning a value of `-1`. A valid data set name is then provided and the returned index is used to read the contents of the corresponding data set.

C:

```
#include "hdf.h"
#include "mfhdf.h"

#define X_LENGTH 4
#define Y_LENGTH 5
#define Z_LENGTH 6

main( )
{

int32 sd_id, sds_index, sds_id, status;
int32 start[3], edges[3];
int16 array_data[Z_LENGTH][Y_LENGTH][X_LENGTH];

/* Open the file. */
sd_id = SDstart("Example4.hdf", DFACC_RDONLY);

/* Search for the index of a non-existent array data set. */
sds_index = SDnametoindex(sd_id, "Invalid_Data_Set_Name");

/* Error condition: sds_index contains the value -1. */

/* Search for the index of a "Ex_array_4" array data set. */
sds_index = SDnametoindex(sd_id, "Ex_array_4");

/* Select the data set corresponding to the returned index. */
sds_id = SDselect(sd_id, sds_index);

/* Read the data set data into the array_data array. */
start[0] = start[1] = start[2] = 0;
edges[0] = Z_LENGTH;
edges[1] = Y_LENGTH;
edges[2] = X_LENGTH;

status = SDreaddata(sds_id, start, NULL, edges, (VOIDP)array_data);

/* Terminate access to the array */
```

```
status = SDendaccess(sds_id);

/* Terminate access to the SD interface and close the file */
status = SDend(sd_id);

}
```

FORTRAN:

```
     PROGRAM SEARCH INDEX

      integer*4 sd_id, sds_index, sds_id, status
      integer*4 start(3), edges(3), stride(3)
      integer sfstart, sfn2index, sfrdata, sfendacc, sfselect

C     DFACC_RDONLY is defined in hdf.h.
      integer*4 DFACC_RDONLY
      integer*4 X_LENGTH, Y_LENGTH, Z_LENGTH
      parameter (DFACC_RDONLY = 1, X_LENGTH = 4, Y_LENGTH = 5,
     +           Z_LENGTH = 6)
      integer*2 array_data(X_LENGTH, Y_LENGTH, Z_LENGTH)

C     Open the file.
      sd_id = sfstart('Example4.hdf', DFACC_RDONLY)

C     Search for the index of a non-existent array data set.
      sds_index = sfn2index(sd_id, 'Invalid_Data_Set_Name')

C     Error condition: sds_index contains the value -1.

C     Search for the index of a 'Ex_array_4' array data set.
      sds_index = sfn2index(sd_id, 'Ex_array_4')

C     Select the data set corresponding to the returned index.
      sds_id = sfselect(sd_id, sds_index)

C     Read the data set data into the rd_data array.
      start(1) = 0
      start(2) = 0
      start(3) = 0
      edges(1) = X_LENGTH
      edges(2) = Y_LENGTH
      edges(3) = Z_LENGTH
      stride(1) = 1
      stride(2) = 1
      stride(3) = 1
      status = sfrdata(sds_id, start, stride, edges, array_data)

C     Terminate access to the array
      status = sfendacc(sds_id)

      end
```

3.7.5 Creating SDS Arrays Containing Variable-Length Data: SDsetnbitdataset

Version 4.0r1 of HDF provided the **SDsetnbitdataset** routine, allowing the HDF user to specify that a particular SDS array contains data of a non-standard length. Any length between 1 and 32 bits can be specified. After **SDsetnbitdataset** has been called for the SDS array, any read or write operations will involve a conversion between the new data length of the SDS array and the data length of the read or write buffer.

The syntax of **SDsetnbitdataset** is as follows:

C:

```
status = SDsetnbitdataset(sds_id, start_bit, bit_len,
                          sign_ext, fill_one);
```

FORTRAN:

```
status = sfsnbit(sds_id, start_bit, bit_len, sign_ext,
                 fill_one)
```

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 value of `1` would denote 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 is set to either `TRUE` or `FALSE` - specify `TRUE` to sign-extend.

The `fill_one` specifies whether to fill the "background" bits with the value `1` or `0`. This parameter is also set to either `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.

SDsetnbitdataset returns `SUCCEED` (or `0`) upon successful completion and `FAIL` (or `-1`) otherwise.

TABLE 3L

SDsetnbitdataset Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDsetnbitdataset (sfsnbit)	sds_id	int32	integer	Data set identifier.
	start_bit	intn	integer	Leftmost bit of the field to be written.
	bit_len	intn	integer	Length of the bit field to be written
	sign_ext	intn	integer	Sign-extend specifier.
	fill_one	intn	integer	Background bit specifier.

3.8 Chunked (or Tiled) Scientific Data Sets

NOTE: It is strongly encouraged that HDF users who wish to use the SD chunking routines first read the section on SD chunking in Chapter 13, titled *HDF Performance Issues*. In that section the concepts of chunking are explained, as well as their use in relation to HDF. As the ability to work with chunked data has been added to HDF functionality for the purpose of addressing specific performance-related issues, the user should first have the necessary background knowledge to correctly determine how chunking will positively or adversely affect their application.

This chapter will refer to both "tiled" and "chunked" SDSs as simply "chunked SDSs", as tiled SDSs are the two-dimensional case of chunked SDSs.

3.8.1 Making a Non-Chunked SDS a Chunked SDS: SDsetchunk

In HDF, an SDS must first be created as a generic SDS through the **SDcreate** routine, then "promoted" to be a chunked SDS through the use of **SDsetchunk**. **SDsetchunk** determines the chunk size and the compression method, if any, to be applied when accessing the chunks.

SDsetchunk receives its information on how to create the chunks, as well as compression information, from an `HDF_CHUNK_DEF` union passed in as its second argument. This union structure is defined in the HDF library as follows:

```
typedef union hdf_chunk_def_u {
    int32 chunk_lengths[MAX_VAR_DIMS];
    struct {
        int32 chunk_lengths[MAX_VAR_DIMS];
        int32 comp_type;
        comp_info cinfo;
    } comp;
}HDF_CHUNK_DEF
```

Note that there is currently no Fortran-77 version of **SDsetchunk**.

TABLE 3M **SDsetchunk Parameter List**

Routine Name	Parameter	Data Type	Description
		C	
SDsetchunk	sds_id	int32	SD identifier.
	c_def	HDF_CHUNK_DEF	Union containing information on how the chunks are to be defined.
	flags	int32	Flags determining the behavior of the routine.

The `flags` parameter can either be set to `HDF_CHUNK` if the SDS is to be uncompressed, or to the bitwise-OR'ed values of `HDF_CHUNK` and `HDF_COMP` (`HDF_CHUNK | HDF_COMP`) if a compression method is to be applied. In the former case the first definition of the `chunk_lengths` array should be set to the dimensions of the chunks. For example, given the definition of `HDF_CHUNK_DEF` stated above, a `flags` parameter value of `HDF_CHUNK` and the following definition of a three-dimensional chunked SDS

```
HDF_CHUNK_DEF current_chunk_def;

current_chunk_def.chunk_lengths[0] = 2;
current_chunk_def.chunk_lengths[1] = 3;
current_chunk_def.chunk_lengths[2] = 4;
```

the size of the chunk passed to **SDsetchunk** will be 2 bytes by 3 bytes by 4 bytes.

In the latter case (where a compression method is specified), the `chunk_lengths` array definition within the `comp` structure is initialized in the way described above and the `cinfo` structure is initialized in the same way as for the **SDsetcompress** routine. (See Figure 3.5.3 on page 40.)

There are two restrictions that apply to chunked SDSs. The maximum number of chunks in a single HDF file is 65,535, and a chunked SDS cannot contain an unlimited dimension.

SDsetchunk returns either a value of `SUCCEED` or `FAIL`.

3.8.2 Setting the Maximum Number of Chunks in the Cache: SDsetchunkcache

To maximize the performance of the HDF library routines when working with chunked SDSs, the library maintains a separate area of memory specifically for cached data chunks. **SDsetchunkcache** determines the maximum number of chunks of the specified SDS that are cached into this segment of memory.

When the chunk cache has been filled, any additional chunks written to cache memory are cached according to the LRU, or Least-Recently-Used, algorithm. This means that the chunk that has resided in the cache the longest without being reread or rewritten will be written over with the new chunk.

Note that there is currently no Fortran-77 equivalent of **SDsetchunkcache**.

TABLE 3N **SDsetchunkcache Parameter List**

Routine Name	Parameter	Data Type	Description
		C	
SDsetchunkcache	sds_id	int32	SD identifier.
	maxcache	int32	Maximum number of chunks to cache.
	flags	int32	Flags determining the default caching behavior.

By default, when a generic SDS is promoted to be a chunked SDS, the `maxcache` parameter is set to the number of chunks along the last dimension.

If the chunk cache is full and the value of the `maxcache` parameter is larger than the currently allowed maximum number of cached chunks, then the maximum number of cached chunks is reset to the value of `maxcache`. If the chunk cache is not full, then the size of the chunk cache is reset to the value of `maxcache` only if it is greater than the current number of chunks in the cache.

Currently the only allowed value of the flag parameter is `0`, which designates default operation. In the near future, the value `HDF_CACHEALL` will be supported to be used to specify that the whole SDS object is to be cached.

SDsetchunkcache returns either the set value of the maximum number of cacheable chunks or `FAIL`.

3.8.3 Writing Data to Chunked SDSs: SDwritechunk and SDwritedata

If an SDS has been created as a chunked SDS (i.e., has been created through calls to **SDcreate** and **SDsetchunk**), both **SDwritedata** and **SDwritechunk** can be used to write data to it. There are situations where **SDwritechunk** may be a more appropriate routine to use than **SDwritedata**, but both routines essentially achieve the same results.

The location of data in a chunked SDS can be referenced in two ways. The first is the standard method used in the SD routines that access both chunked and non-chunked SDSs, and refers to the starting location, or the *origin*, as an offset in bytes from the origin of the SD array itself. The second method is used by the SD routines that only access chunked SDSs, and refers to the origin of the chunk as an offset in chunks from the origin of the SD array itself. See the section on SD chunking in Chapter 13, titled *HDF Performance Issues*, for an illustration of this.

SDwritechunk is used when an entire chunk is to be written and requires that the chunk offset be known. **SDwritedata** is used when the write operation is to be done regardless of the chunking scheme used in the SDS and the byte offset of the target chunk is known. Also, as **SDwritechunk** is written specifically for chunked SDSs and doesn't have the overhead of the additional functionality supported by the **SDwritedata** routine, it is much faster than **SDwritedata**.

The parameters of **SDwritedata** are listed above. (See Table 3D on page 30.) The parameters of **SDwritechunk** are as follows. Note that there currently is no Fortran-77 equivalent of **SDwritechunk**.

TABLE 3O **SDwritechunk Parameter List**

Routine Name	Parameter	Data Type	Description
		C	
SDwritechunk	sds_id	int32	SD identifier.
	origin	int32 *	Origin of the chunk to be written.
	datap	const VOID *	Buffer containing the data to be written.

The `datap` parameter must point to an array containing an entire chunk of data - in other words, the size of the array must be the same as the chunk size of the SDS to be written to. An error condition will result if this is not the case.

SDwritechunk returns either a value of `SUCCEED` or `FAIL`.

3.8.4 Reading Data From Chunked SDSs: SDreadchunk and SDreaddata

As both **SDwritedata** and **SDwritechunk** can be used to write data to chunked SDSs, both **SDreaddata** and **SDreadchunk** can be used to read data from chunked SDSs.

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 SDS. Also, **SDreadchunk** is written specifically for chunked SDSs and doesn't have the overhead of the additional functionality supported by the **SDreaddata** routine - therefore, it is much faster than **SDreaddata**.

SDreadchunk returns either a value of `SUCCEED` or `FAIL`.

The parameters of **SDreaddata** are listed above. (See Table 3I on page 47.) The parameters of **SDreadchunk** are as follows. Note that there is currently no Fortran-77 equivalent of **SDreadchunk**.

TABLE 3P

SDreadchunk Parameter List

Routine Name	Parameter	Data Type	Description
		C	
SDreadchunk	sds_id	int32	SD identifier.
	origin	int32 *	Origin of the chunk to be read.
	datap	VOID *	Buffer for the returned chunk data.

As with **SDwritechunk**, the `datap` parameter must point to an array containing enough space for an entire chunk of data. In other words, the size of the array must be the same as the chunk size of the SDS to be written to. An error condition will result if this is not the case.

3.8.5 Obtaining Information About a Chunked SDS: SDgetchunkinfo

SDgetchunkinfo is used to determine how the chunks in a chunked SDS are defined and whether the SDS is chunked or not.

Information about the chunks is returned in the `HDF_CHUNK_DEF` union provided as the second parameter - therefore, it will return the same information that can be set in the **SDsetchunk** routine. For example, the `flags` argument will return either `HDF_CHUNK` for an uncompressed chunked SDS, `HDF_CHUNK` bitwise-OR'ed with `HDF_COMP` for a compressed and chunked SDS or `HDF_NONE` for a non-chunked SDS. A pointer to a `chunk_lengths` array containing chunk dimension size information will be returned if the returned `flags` param value is `HDF_CHUNK`, a pointer to a `comp` structure containing chunk dimension size and compression information will be returned if the returned `flags` param value is `HDF_CHUNK` bitwise-OR'ed with `HDF_COMP`.

A `NULL` value can also be passed in as the `c_def` param if chunking information is not desired.

Note that there is currently no Fortran-77 equivalent of **SDgetchunkinfo**.

TABLE 3Q

SDgetchunkinfo Parameter List

Routine Name	Parameter	Data Type	Description
		C	
SDgetchunkinfo	sds_id	int32	SD identifier.
	c_def	HDF_CHUNK_DEF *	Union structure containing information about the chunks in the SDS.
	flags	int32 *	Flags determining the behavior of the routine.

SDgetchunkinfo returns either a value of `SUCCEED` or `FAIL`.

EXAMPLE 13.

Writing and Reading Chunked Data Using SDwritechunk and SDreaddata

This example creates a 9-by-4 integer chunked uncompressed SDS in a file named "Example13.hdf", then writes six 3-by-2 byte chunks of 16-bit unsigned integers to it. It then reads one 5-by-2 16-bit unsigned integer subset. It uses **SDwritechunk** to write the chunks and **SDreaddata** to read the subset.

C:

```
#include "hdf.h"
#include "mfhdf.h"

/* Arrays containing dimension info for datasets. */
static int32  d_dims[3]     = {2, 3, 4};  /* Data dimensions */
static int32  edge_dims[3]  = {0, 0, 0};  /* Edge dims */
static int32  start_dims[3] = {0, 0, 0};  /* Starting dims  */
static int32  cdims[3]      = {1, 2, 3};  /* Chunk lengths */

int32 status;

static uint16  chunk1_2u16[6] = {11, 21,
                                 12, 22,
                                 13, 23};

static uint16  chunk2_2u16[6] = {31, 41,
                                 32, 42,
                                 33, 43};

static uint16  chunk3_2u16[6] = {14, 24,
                                 15, 25,
                                 16, 26};

static uint16  chunk4_2u16[6] = {34, 44,
                                 35, 45,
                                 36, 46};

static uint16  chunk5_2u16[6] = {17, 27,
                                 18, 28,
                                 19, 29};

static uint16  chunk6_2u16[6] = {37, 47,
                                 38, 48,
                                 39, 49};

main( )
{

int32 f1;       /* File handle */
int32 sdsid;/* SDS handle */
uint16  inbuf_2u16[5][2];/* Data array read */
uint16  fill_u16 = 0;/* Fill value */
int32 c_flags;

HDF_CHUNK_DEF c_def, r_def;

/* Create the HDF file. */
f1 = SDstart("Example13.hdf", DFACC_CREATE);

/* Create a  9x4 SDS of uint16 in file 1. */
d_dims[0] = 9;
d_dims[1] = 4;

sdsid = SDcreate(f1, "DataSetChunked_1", DFNT_UINT16, 2, d_dims);

/* Set the fill value. */
fill_u16 = 0;
status = SDsetfillvalue(sdsid, (VOIDP) &fill_u16);

/* Create chunked SDS chunk with 3x2 chunks which will create
```

```
6 chunks. */

c_def.chunk_lengths[0] = 3;
c_def.chunk_lengths[1] = 2;

status = SDsetchunk(sdsid, c_def, HDF_CHUNK);

/* Set chunk cache to hold a maximum of 3 chunks */
status = SDsetchunkcache(sdsid, 3, 0);

/* Write the data chunks. */

/* Write chunk 1. */
start_dims[0] = 0;
start_dims[1] = 0;
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk1_2u16);

/* Write chunk 4. */
start_dims[0] = 1;
start_dims[1] = 1;
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk4_2u16);

/* Write chunk 2. */
start_dims[0] = 0;
start_dims[1] = 1;
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk2_2u16);

/* Write chunk 5. */
start_dims[0] = 2;
start_dims[1] = 0;
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk5_2u16);

/* Write chunk 3. */
start_dims[0] = 1;
start_dims[1] = 0;
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk3_2u16);

/* Write chunk 6. */
start_dims[0] = 2;
start_dims[1] = 1;
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk6_2u16);

/* Read a subset of the data back in using SDreaddata
       i.e  5x2 subset of the whole array. */
start_dims[0] = 2;
start_dims[1] = 1;
edge_dims[0] = 5;
edge_dims[1] = 2;
status = SDreaddata(sdsid, start_dims, NULL, edge_dims, \
                (VOIDP) inbuf_2u16);

/* This 5x2 array should look like this
         {{23, 24, 25, 26, 27},
          {33, 34, 35, 36, 37}} */

/* Get chunk lengths. */
status = SDgetchunkinfo(sdsid, &r_def, &c_flags);

/* Close the current SDS. */
status = SDendaccess(sdsid);

/* Close down the SDS interface. */
status = SDend(f1);
```

```
}
```

EXAMPLE 14.

Writing and Reading Chunked Data Using SDwritedata and SDreaddata

This example creates a chunked SDS with a size of 2-by-3-by-4 16-bit unsigned integers in a file named "Example14.hdf", then writes two 2-by-3-by-2 16-bit unsigned integer chunks to it. It then reads one 2-by-3-by-4 16-bit unsigned integer chunk. It uses **SDwritedata** to write the chunks and **SDreaddata** to read the subset.

C:

```
#include "hdf.h"
#include "mfhdf.h"

/* Arrays containing dimension info for datasets. */
static int32  d_dims[3]     = {2, 3, 4};  /* Data dimensions */
static int32  edge_dims[3]  = {0, 0, 0};  /* Edge dims */
static int32  start_dims[3] = {0, 0, 0};  /* Starting dims  */
static int32  cdims[3]      = {1, 2, 3};  /* Chunk lengths */

int32 status;

static uint16  u16_3data[2][3][4] =
{
{
    { 0, 1, 2, 3},
    { 10, 11, 12, 13},
    { 20, 21, 22, 23}},
{
    { 100, 101, 102, 103},
    { 110, 111, 112, 113},
    { 120, 121, 122, 123}}};

main( )
{

int32 f1;/* File handle */
int32 sdsid;/* SDS handle */
uint16  inbuf_3u16[2][3][4];/* Data array read */
uint16  fill_u16 = 0;        /* Fill value */
int32 c_flags;

HDF_CHUNK_DEF c_def, r_def;

/* Create the HDF file. */
f1 = SDstart("Example14.hdf", DFACC_CREATE);

/* Create a new 2x3x4 SDS of uint16 in the file. */
d_dims[0] = 2;
d_dims[1] = 3;
d_dims[2] = 4;
sdsid = SDcreate(f1, "DataSetChunked_2", DFNT_UINT16, 3, d_dims);

/* Set the fill value. */
fill_u16 = 0;
status = SDsetfillvalue(sdsid, (VOIDP) &fill_u16);

/* Create chunked SDS - chunk is 2x3x2 which will create 2 chunks */
c_def.chunk_lengths[0] = 2;
```

```
c_def.chunk_lengths[1] = 3;
c_def.chunk_lengths[2] = 2;
status = SDsetchunk(sdsid, c_def, HDF_CHUNK);

/* Set chunk cache to hold a maximum of 2 chunks. */
status = SDsetchunkcache(sdsid, 2, 0);

/* Write data using SDwritedata. */
start_dims[0] = 0;
start_dims[1] = 0;
start_dims[2] = 0;
edge_dims[0] = 2;
edge_dims[1] = 3;
edge_dims[2] = 4;
status = SDwritedata(sdsid, start_dims, NULL, edge_dims, \
                (VOIDP) u16_3data);

/* Read data using SDreaddata. */
start_dims[0] = 0;
start_dims[1] = 0;
start_dims[2] = 0;
edge_dims[0] = 2;
edge_dims[1] = 3;
edge_dims[2] = 4;
status = SDreaddata(sdsid, start_dims, NULL, edge_dims, \
                (VOIDP) inbuf_3u16);

/* Verify the data in inbuf_3u16 against u16_3data[]. */

/* Get chunk lengths. */
status = SDgetchunkinfo(sdsid, &r_def, &c_flags);

/* Close the current SDS. */
status = SDendaccess(sdsid);

/* Close down SDS interface. */
status = SDend(f1);

}
```

EXAMPLE 15.

Writing and Reading Chunked Data Using SDwritedata and SDreadchunk

This example creates a chunked SDS with a size of 2-by-3-by-4 16-bit unsigned integers in a file named "Example15.hdf", then writes one 2-by-3-by-4 16-bit unsigned integer chunk to it. It then reads six 1-by-1-by-4 16-bit unsigned integer subsets. It uses **SDwritedata** to write the chunks and **SDreadchunk** to read the subsets.

C:

```
#include "hdf.h"
#include "mfhdf.h"

/* Arrays containing dimension info for datasets. */
static int32  d_dims[3]     = {2, 3, 4};  /* Data dimensions */
static int32  edge_dims[3]  = {0, 0, 0};  /* Edge dims */
static int32  start_dims[3] = {0, 0, 0};  /* Starting dims  */
static int32  cdims[3]      = {1, 2, 3};  /* Chunk lengths */

static uint16  u16_3data[2][3][4] =
{
```

```
{
    { 0, 1, 2, 3},
    { 10, 11, 12, 13},
    { 20, 21, 22, 23}},
{
    { 100, 101, 102, 103},
    { 110, 111, 112, 113},
    { 120, 121, 122, 123}}};

main( )
{

int32 f1;       /* File handle */
int32 sdsid;/* SDS handle */
uint16  ru16_3data[4];/* Whole chunk input buffer */
int32   rcdims[3];/* For SDgetchunkinfo() */
uint16  fill_u16 = 0;/* Fill value */
int32 c_flags;
HDF_CHUNK_DEF c_def, r_def;

int32 status;

/* Create the HDF file. */
f1 = SDstart("Example15.hdf", DFACC_CREATE);

/* Create a new 2x3x4 SDS of uint16 in the file. */
d_dims[0] = 2;
d_dims[1] = 3;
d_dims[2] = 4;
sdsid = SDcreate(f1, "DataSetChunked_4", DFNT_UINT16, 3, d_dims);

/* Set the fill value. */
fill_u16 = 0;
status = SDsetfillvalue(sdsid, (VOIDP) &fill_u16);

/* Create chunked SDS - chunk is 1x1x4 which will create 6 chunks. */
c_def.chunk_lengths[0] = 1;
c_def.chunk_lengths[1] = 1;
c_def.chunk_lengths[2] = 4;
status = SDsetchunk(sdsid, c_def, HDF_CHUNK);

/* Set chunk cache to hold a maximum of 4 chunks. */
status = SDsetchunkcache(sdsid, 4, 0);

/* Write data using SDwritedata. */
start_dims[0] = 0;
start_dims[1] = 0;
start_dims[2] = 0;
edge_dims[0] = 2;
edge_dims[1] = 3;
edge_dims[2] = 4;
status = SDwritedata(sdsid, start_dims, NULL, edge_dims, \
                        (VOIDP) u16_3data);

/* Read data using SDreadchunk and verify against
       the chunk arrays chunk1_3u16[] ... chunk6_3u16[]. */

/* Read chunk 1. */
start_dims[0] = 0;
start_dims[1] = 0;
start_dims[2] = 0;
status = SDreadchunk(sdsid, start_dims, (VOIDP) ru16_3data);
```

```
/* Read chunk 2. */
start_dims[0] = 0;
start_dims[1] = 1;
start_dims[2] = 0;
status = SDreadchunk(sdsid, start_dims, (VOIDP) ru16_3data);

/* Read chunk 3. */
start_dims[0] = 0;
start_dims[1] = 2;
start_dims[2] = 0;
status = SDreadchunk(sdsid, start_dims, (VOIDP) ru16_3data);

/* Read chunk 4. */
start_dims[0] = 1;
start_dims[1] = 0;
start_dims[2] = 0;
status = SDreadchunk(sdsid, start_dims, (VOIDP) ru16_3data);

/* Read chunk 5. */
start_dims[0] = 1;
start_dims[1] = 1;
start_dims[2] = 0;
status = SDreadchunk(sdsid, start_dims, (VOIDP) ru16_3data);

/* Read chunk 6. */
start_dims[0] = 1;
start_dims[1] = 2;
start_dims[2] = 0;
status = SDreadchunk(sdsid, start_dims, (VOIDP) ru16_3data);

/* Get chunk lengths. */
status = SDgetchunkinfo(sdsid, &r_def, &c_flags);

/* Close the current SDS. */
status = SDendaccess(sdsid);

/* Close down the SDS interface. */
status = SDend(f1);

}
```

EXAMPLE 16.

Writing and Reading Compressed Chunked Data Using SDwritechunk and SDreaddata

This example uses **SDwritechunk** to write the chunks and **SDreaddata** to read the subset, like Example 13 (See page 62.). However, it also compresses the chunks using the GZIP (skipping Huffman) algorithm.

C:

```
#include "hdf.h"
#include "mfhdf.h"

/* Arrays containing dimension info for datasets. */
static int32  d_dims[3]     = {2, 3, 4};  /* Data dimensions */
static int32  edge_dims[3]  = {0, 0, 0};  /* Edge dims */
static int32  start_dims[3] = {0, 0, 0};  /* Starting dims  */
static int32  cdims[3]      = {1, 2, 3};  /* Chunk lengths */

int32 status;

static uint16  chunk1_2u16[6] = {11, 21,
```

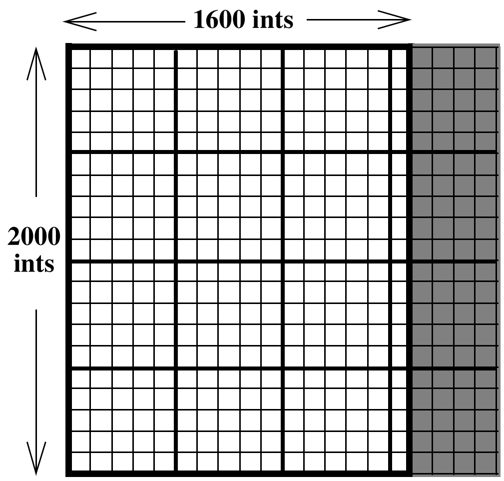
```
                                      12, 22,
                                      13, 23};

static uint16  chunk2_2u16[6] = {31, 41,
                                      32, 42,
                                      33, 43};

static uint16  chunk3_2u16[6] = {14, 24,
                                      15, 25,
                                      16, 26};

static uint16  chunk4_2u16[6] = {34, 44,
                                      35, 45,
                                      36, 46};

static uint16  chunk5_2u16[6] = {17, 27,
                                      18, 28,
                                      19, 29};

static uint16  chunk6_2u16[6] = {37, 47,
                                      38, 48,
                                      39, 49};

main( )
{

int32 f1;       /* File handle */
int32 sdsid;/* SDS handle */
uint16  inbuf_2u16[5][2];    /* Data array read */
uint16  fill_u16 = 0;        /* Fill value */
int32 c_flags;
HDF_CHUNK_DEF c_def, r_def;

/* Create the HDF file. */
f1 = SDstart("Example16.hdf", DFACC_CREATE);

/* Create a 9x4 SDS of uint16 in file 1. */
d_dims[0] = 9;
d_dims[1] = 4;
sdsid = SDcreate(f1, "DataSetChunked_1", DFNT_UINT16, 2, d_dims);

/* Set the fill value. */
fill_u16 = 0;
status = SDsetfillvalue(sdsid, (VOIDP) &fill_u16);

/* Create chunked SDS chunk with 3x2 chunks which will create
    6 chunks. */
c_def.chunk_lengths[0] = c_def.comp.chunk_lengths[0] = 3;
c_def.chunk_lengths[1] = c_def.comp.chunk_lengths[1] = 2;

c_def.comp.comp_type = COMP_CODE_DEFLATE; /* GZIP */
c_def.comp.cinfo.deflate.level = 6;

status = SDsetchunk(sdsid, c_def, HDF_CHUNK | HDF_COMP);

/* Set chunk cache to hold a maximum of 3 chunks */
status = SDsetchunkcache(sdsid, 3, 0);

/* Write the data chunks. */

/* Write chunk 1. */
start_dims[0] = 0;
start_dims[1] = 0;
```

```
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk1_2u16);

/* Write chunk 4. */
start_dims[0] = 1;
start_dims[1] = 1;
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk4_2u16);

/* Write chunk 2. */
start_dims[0] = 0;
start_dims[1] = 1;
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk2_2u16);

/* Write chunk 5. */
start_dims[0] = 2;
start_dims[1] = 0;
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk5_2u16);

/* Write chunk 3. */
start_dims[0] = 1;
start_dims[1] = 0;
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk3_2u16);

/* Write chunk 6. */
start_dims[0] = 2;
start_dims[1] = 1;
status = SDwritechunk(sdsid, start_dims, (VOIDP) chunk6_2u16);

/* Read a subset of the data back in using SDreaddata
       i.e  5x2 subset of the whole array. */
start_dims[0] = 2;
start_dims[1] = 1;
edge_dims[0] = 5;
edge_dims[1] = 2;
status = SDreaddata(sdsid, start_dims, NULL, edge_dims, \
                (VOIDP) inbuf_2u16);

/* This 5x2 array should look like this
         {{23, 24, 25, 26, 27},
          {33, 34, 35, 36, 37}} */

/* Get chunk lengths. */
status = SDgetchunkinfo(sdsid, &r_def, &c_flags);

/* Close the current SDS. */
status = SDendaccess(sdsid);

/* Close down the SDS interface. */
status = SDend(f1);

}
```

3.8.6 Ghost Areas

In cases where the size of the SDS array is not an even multiple of the chunk size, regions of excess array space beyond the defined dimensions of the SDS will be created. Refer to the following illustration.

FIGURE 3c **Array Locations Created Beyond the Defined Dimensions of an SDS**



In a 1600 by 2000 integer chunked SDS array with 500 by 500 integer chunks, a 400 by 2000 integer area of array locations beyond the defined dimensions of the SDS is created (shaded area). These areas are called "ghost areas".

These regions are called "ghost areas". "Ghost areas" can be accessed only by **SDreadchunk** and **SDwritechunk** - they cannot be written to or accessed by either **SDreaddata** or **SDwritedata**. If the fill value has been set, the values in these array locations will be initialized to the fill value. It is highly recommended that users set the fill value before writing to chunked SDSs so that garbage values won't be read from these locations.

Also, if **SDreadchunk** and **SDwritechunk** are used, it is not recommended that valid data be written to the "ghost areas" as these areas won't be accessible by **SDreaddata** and **SDwritedata**. Moreover, the ability to write to these areas may not be supported in future versions of the HDF library.

3.9 SD Dimension and Dimension Scale Operations

A description of dimensions is in Section 3.2.1.4 on page 20.

3.9.1 Selecting a Dimension: SDgetdimid

SDS dimensions are uniquely identified by ***dimension ids***, which are assigned when a dimension is created. These dimension ids are used to refer within a program to a particular dimension, its scale and its attributes. Before working with a dimension, a program must first obtain a dimension id by calling the **SDgetdimid** routine:

C:
```
dim_id = SDgetdimid(sds_id, dim_index);
```

FORTRAN:
```
dim_id = sfdimid(sds_id, dim_index)
```

SDgetdimid has two arguments; `sds_id` and `dim_index`, and returns a dimension identifier, `dim_id`. The argument `dim_index` is a zero-based integer indicating the location of the dimension in the data set.

The number of dimensions in a data set is specified at the time the data set is created. Specifying a dimension index larger than the number of dimensions in the data set returns an error.

Unlike file and data set identifiers, dimension identifiers do not require explicit disposal.

3.9.2 Naming a Dimension: SDsetdimname

SDsetdimname assigns a name to the selected dimension. The name of the dimension will also appear as the name of the dimension scale. If the dimension name is not unique, it is assumed that both dimensions refer to the same object and changes to one will be reflected in the other. Naming dimensions is optional but encouraged. Dimensions that are not explicitly named by the user will have names generated by the HDF library. Use **SDdiminfo** to read existing dimension names.

The following steps are required to name a dimension and its scale:

1. Get the identifier of the dimension.
2. Assign a name to the dimension - the dimension scale will be set automatically.

The following routine calls are required to do this:

C:
```
dim_id = SDgetdimid(dim_id, dim_index);
status = SDsetdimname(dim_id, dim_name);
```

FORTRAN:
```
dim_id = sfdimid(dim_id, dim_index)
status = sfsdimname(dim_id, dim_name)
```

The argument `dim_id` in **SDsetdimname** is the dimension identifier returned by **SDgetdimid** and `dim_name` is the name for the selected dimension. An attempt to rename a dimension using **SDsetdimname** will cause the old name to be deleted and a new one to be assigned.

What should be remembered when naming dimensions is that the name or a particular dimension *must* be set before attributes are assigned - and, once the attributes have been set, the name should not be changed. In other words, **SDsetdimname** should only be called before any calls to **SDsetdimscale** (described in Section 3.9.4.1 on page 74), **SDsetattr** (described in Section 3.10.1 on page 81) or **SDsetdimstrs** (described in Section 3.11.2.1 on page 89).

TABLE 3R **SDsetdimname Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDsetdimname (sfsdimname)	dim_id	int32	integer	Dimension identifier.
	dim_name	char *	character* (*)	Dimension name.

3.9.3 Old and New Dimension Implementations

Up to and including HDF version 4.0 beta1, dimensions were vgroup objects containing a single field vdata with a class name of "DimVal0.0". The vdata had the same number of records as the size of the dimension, which consisted of the values 0, 1, 2, . . . n - 1, where n is the size of the dimension. These values weren't strictly necessary and for applications that create large one dimensional array datasets the disk space taken by these unnecessary values would nearly double the size of the HDF file. In order to avoid these situations, a new representation of dimensions was implemented for HDF version 4.0 beta 2 and later versions.

Dimensions are still vgroups in the new representation: the only differences are that the vdata has only one record with a value of `<dimension size>` and the class name of the vdata has been changed to "DimVal0.1" to distinguish it from the old version.

Until HDF version 4.1, the old and new dimension representation will be written by default for each dimension created, and both representations will be recognized by routines that operate on dimensions. HDF version 4.1 routines will only recognize the new representation. During the transitional period, two routines will be provided to allow HDF programs to distinguish between the two dimension representations, or *compatibility modes*: - **SDsetdimval_comp** and **SDsetdimval_bwcomp**.

3.9.3.1 Setting the Future Compatibility Mode of a Dimension: SDsetdimval_comp

SDsetdimval_comp determines whether the specified dimension *will have* the old and new representations or the new representation only, by setting the compatibility mode for the specified dimension. The routine's syntax is the following:

C:
```
status = SDsetdimval_comp(dim_id, comp_mode);
```

FORTRAN:
```
status = sfsdmvc(dim_id, comp_mode)
```

The `comp_mode` parameter determines the compatibility mode. It can be set to either `SD_DIMVAL_BW_COMP`, which specifies compatible mode and that the old and new dimension representations will be written to file , or `SD_DIMVAL_BW_INCOMP`, which specifies incompatible mode and that only the new dimension representation will be written to file.

Unlimited dimensions are always backward compatible. Therefore, **SDsetdimval_comp** takes no action on these dimensions.

SDsetdimval_comp returns either `SUCCEED` on successful completion, or `FAIL` otherwise.

TABLE 3S **SDsetdimval_comp Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDsetdimval_comp (sfsdmvc)	dim_id	int32	integer	Dimension identifier.
	comp_mode	intn	integer	Compatibility mode.

3.9.3.2 Setting the Current Compatibility Mode of a Dimension: SDisdimval_bwcomp

SDisdimval_bwcomp determines whether the specified dimension *has* the old and new representations or the new representation only. The function's syntax is the following:

C:
```
comp_mode = SDisdimval_bwcomp(dim_id);
```

FORTRAN:
```
comp_mode = sfisdmvc(dim_id)
```

SDisdimval_bwcomp returns one of three values: either `SD_DIMVAL_BW_COMP`, `SD_DIMVAL_BW_INCOMP` or `ERROR`. `SD_DIMVAL_BW_COMP` and `SD_DIMVAL_BW_INCOMP` are interpreted as they are by **SDisdimval_comp**.

TABLE 3T

SDisdimval_bwcomp Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDisdimval_bwcomp (sfisdmvc)	dim_id	int32	integer	Dimension identifier.

3.9.4 Dimension Scales

A dimension scale is a series of numbers placed along a dimension to demarcate intervals in a data set. One scale is assigned per dimension. In the SDS data model, each dimension scale is a one-dimensional array with size and name equal to its assigned dimension name and size. For example, if a dimension of length 6 named "depth" is assigned a dimension scale, its scale is a one-dimensional array of length 6 and is also assigned the name "depth".

Although dimension scales are conceptually different from SDS arrays, they are both arrays and are often treated in the same way by the SDS API. For example, when the **SDfileinfo** routine returns the number of data sets in a file, it includes dimension scales in that number. The **SDiscoordvar** routine (described in Section 3.9.5 on page 78) distinguishes data sets from dimension scales.

3.9.4.1 Writing Dimension Scales: SDsetdimscale

Selecting a dimension with **SDgetdimid** assigns a default dimension scale. Default dimension scales have the data type of the corresponding SDS array. To assign a non-default scale with a non-default data type, use **SDsetdimscale**.

To create a non-default scale, the following steps are required:

1. Get the identifier of the dimension.
2. Create the dimension scale, setting the data type and size to the desired values.

To do this, the calling program must execute the following calls:

C:
```
dim_id = SDgetdimid(sds_id, dim_index);
status = SDsetdimscale(dim_id, count, data_type, data);
```

FORTRAN:
```
dim_id = sfdimid(sds_id, dim_index)
status = sfsdscale(dim_id, count, data_type, data)
```

The argument `count` is the size of the scale, `data_type` defines the data type for the scale values and `data` is an array containing the scale values. Assigning a value to `count` is optional: it exists to insure backward compatibility.

3.9.4.2 Obtaining Dimension Scale and Other Dimension Information: SDdiminfo

Before working with an existing dimension scale, it is often necessary to determine its characteristics. For instance, to allocate the proper amount of memory for a scale requires knowledge of its size and data type. **SDdiminfo** provides this basic information, as well as the name and attribute count for a specified dimension.

To obtain dimension information the following steps are required:

1. Get the identifier of the dimension.
2. Retrieve the dimension information.

The calling program must call the following routines in order:

C:
```
dim_id = SDgetdimid(sds_id, dim_index);
status = SDdiminfo(dim_id, name, count, data_type, nattrs);
```

FORTRAN:
```
dim_id = sfdimid(sds_id, dim_index)
gstatus = sfgdinfo(dim_id, name, count, data_type, nattrs)
```

In **SDdiminfo** the arguments `name`, `count`, `data_type` and `nattrs` define buffers allocated to respectively hold the dimension name and size, the data type for the scale values and the number of attributes assigned to the dimension.

Dimensions are always named. However, if you don't wish to explicitly provide a name, `NULL` can be passed as the `name` parameter to **SDdiminfo** and a default name will be assigned by the SD API. If scale information is available for the dimension, `data_type` will contain the data type of the scale values, otherwise `data_type` will be `0`.

3.9.4.3 Reading Dimension Scales: SDgetdimscale

To read a scale, the following steps are required:

1. Get the identifier of the dimension.
2. Read the scale.

The calling program must contain the following sequence of calls:

C:
```
dim_id = SDgetdimid(sds_id, dim_index);
status = SDgetdimscale(dim_id, data);
```

FORTRAN:
```
dim_id = sfdimid(sds_id, dim_index)
status = sfgdscale(dim_id, data)
```

In **SDgetdimscale** the argument `data` is the buffer allocated to hold the scale values. As **SDgetdimscale** returns all of the values associated with a given scale, it is assumed that the size of the scale buffer is greater than or equal to the dimension size.

TABLE 3U **SDgetdimid, SDsetdimname, SDsetdimscale, SDdiminfo and SDgetdimscale Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDgetdimid (sfdimid)	sds_id	int32	integer	Data set identifier.
	dim_index	intn	integer	Index of the dimension.
SDsetdimname (sfsdmname)	dim_id	int32	integer	Dimension identifier.
	dim_name	char *	character* (*)	Dimension name.
SDsetdimscale (sfsdscale)	dim_id	int32	integer	Dimension identifier.
	count	int32 *	integer	Number of scale values.
	data_type	int32 *	integer	Data type of the scale values.
	data	VOIDP	<valid numeric data type>	Buffer for the scale values.
SDdiminfo (sfgdinfo)	dim_id	int32	integer	Dimension identifier.
	name	char *	character* (*)	Buffer for the dimension name.
	count	int32 *	integer	Buffer for the dimension size.
	number_type	int32 *	integer	Buffer for the scale data type.
	nattrs	int32 *	integer	Buffer for the attribute count.
SDgetdimscale (sfgdscale)	dim_id	int32	integer	Dimension identifier.
	data	VOIDP	<valid numeric data type>	Buffer for the scale values.

EXAMPLE 17.

Writing Dimension Information

The dimensions of the SDS created in Example 4 are created and scales assigned to them in these examples.

C:

```
#include "hdf.h"
#include "mfhdf.h"

main( )
{
int32 sd_id, sds_id, dim_index, dim_id, sds_index, status;
int32 count, num_type, num_attrs;
int16 dim_scale[] = {6,5,4,3,2,1};
char dim_name[MAX_NC_NAME];

/* Open the file. */
sd_id = SDstart("Example4.hdf", DFACC_RDWR);

/* Get the index of the "Ex_array_4" array data set. */
sds_index = SDnametoindex(sd_id, "Ex_array_4");

/* Select the data set corresponding to the returned index. */
sds_id = SDselect(sd_id, sds_index);

/* For each dimension of the "Ex_array_4" array data set, */
for (dim_index = 0; dim_index < 3; dim_index++) {

/* - select the dimension id, */
    dim_id = SDgetdimid(sds_id, dim_index);

/* - get the information about the selected dimension, */
    status = SDdiminfo(dim_id, dim_name, &count, &num_type, \
                       &num_attrs);
    num_type = DFNT_INT16;

/* - alter the dimension names, */
    switch(dim_index) {
        case 0:SDsetdimname(dim_id, "Z_Axis");
                break;
        case 1:SDsetdimname(dim_id, "Y_Axis");
                break;
        case 2:SDsetdimname(dim_id, "X_Axis");
                break;
        default:break;
    }

/* - then alter the dimension scale and write it to the data set. */
    dim_scale[0] = 3;
    dim_scale[1] = 2;
    dim_scale[2] = 1;
    status = SDsetdimscale(dim_id, count, num_type, (VOIDP)dim_scale);
}

/* Terminate access to the array. */
status = SDendaccess(sds_id);

/* Terminate access to the SD interface and close the file. */
status = SDend(sd_id);

}
```

FORTRAN:

```
      PROGRAM ALTER DIMENSION

       integer*4 sd_id, sds_id, dim_index, dim_id, status
       integer*4 count, num_attrs
       integer sfstart, sfn2index, sfdimid, sfgdinfo
       integer sfsdscale, sfsdmname, sfendacc
       integer sfend, sfselect, num_type, i, dim_scale(6)
       integer DFACC_RDWR, MAX_NC_NAME, DFNT_INT16
       parameter (DFACC_RDWR = 3, MAX_NC_NAME = 256, DFNT_INT16 = 22)
       character dim_name(MAX_NC_NAME)

C      For each dimension of the 'Ex_array_4' array data set,
       do 5 i = 1, 6
          dim_scale(i) = i
5      continue

C      Open the file.
       sd_id = sfstart('Example4.hdf', DFACC_RDWR)

C      Get the index of the 'Ex_array_4' array data set.
       sds_index = sfn2index(sd_id, 'Ex_array_4')

C      Select the data set corresponding to the returned index.
       sds_id = sfselect(sd_id, sds_index)

C      For each dimension of the 'Ex_array_4' array data set,
       do 10 dim_index = 1, 3

C       - select the dimension id,
          dim_id = sfdimid(sds_id, dim_index-1)

C       - get the information about the selected dimension,
          status = sfgdinfo(dim_id, dim_name, count, num_type,
      +                     num_attrs)

C       - alter the dimension names.
          if (dim_index .eq. 1) then
          status = sfsdmname(dim_id, 'Z_Axis')
          end if
          if (dim_index .eq. 2) then
          status = sfsdmname(dim_id, 'Y_Axis')
          end if
          if (dim_index .eq. 3) then
          status = sfsdmname(dim_id, 'X_Axis')
          end if

          num_type = DFNT_INT16

C      - and, alter the dimension scale, write it to the data set,
          dim_scale(1) = 3
          dim_scale(2) = 2
          dim_scale(3) = 1
          dim_scale(4) = 0
          dim_scale(5) = -1
          dim_scale(6) = -2
          status = sfsdscale(dim_id, count, num_type, dim_scale)
10     continue

C      Terminate access to the array.
       status = sfendacc(sds_id)
```

```
C     Terminate access to the SD interface and close the file.
      status = sfend(sd_id)
      end
```

3.9.5 Distinguishing SDS Arrays from Dimension Scales: SDiscoordvar

Although dimension scales, or ***coordinate variables*** in netCDF, can for the most part be ignored, it is important to note that, in HDF, they are also SDSs and are assigned the same tag (`DFTAG_SD`) as other SDSs. As a result, dimension scales are treated as SDSs, and are included in the SDS count returned by **SDfileinfo**. The function **SDiscoordvar** is available to determine whether or not a given SDS is a dimension scale.

SDiscoordvar takes an SDS id as its only argument and returns `TRUE` if the data set is a dimension scale, and `FALSE` if it isn't. If **SDiscoordvar** returns `TRUE`, a subsequent call to **SDgetinfo** will fill the specified locations with information about the dimension scale, rather than the data set array.

TABLE 3V **SDiscoordvar Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDiscoordvar (sfiscvar)	sds_id	int32	integer	Data set identifier.

EXAMPLE 18. **Retrieving SDS Information from an HDF File**

SDgetinfo and **SDiscoordvar** provide information that may be needed in order to read in an array successfully, as the following examples show.

C:

```
#include "hdf.h"
#include "mfhdf.h"

#define X_LENGTH 5
#define Y_LENGTH 16

main( )
{

int32 sd_id, sds_id, status;
int32 rank, nt, dims[MAX_VAR_DIMS], nattrs;
int32 start[2], edges[2];
int16 array_data[Y_LENGTH][X_LENGTH];
char name[MAX_NC_NAME];

/* Open the file and initiate the SD interface. */
sd_id = SDstart("Example3.hdf", DFACC_RDONLY);

/* Select the first (and in this case, only) data set in the file. */
sds_id = SDselect(sd_id, 0);

/* Confirm that the data set is not a coordinate variable. */
if (FALSE == SDiscoordvar(sds_id)) {

/* Verify the characteristics of the array. */
    status = SDgetinfo(sds_id, name, &rank, dims, &nt, &nattrs);
```

```
/* Define the location, pattern, and size of the data to read from \
    the data set. */
    start[0] = start[1] = 0;
    edges[0] = dims[0];
    edges[1] = dims[1];

/* Read the array data set created in Example 3. */
    status = SDreaddata(sds_id, start, NULL, edges, (VOIDP)array_data);

}

/* Terminate access to the array data set. */
status = SDendaccess(sds_id);

/* Terminate access to the SD interface and close the file. */
status = SDend(sd_id);

}
```

FORTRAN:

```
      PROGRAM CONFIRM FILE

       integer*4 sd_id, sds_id, rank, nt, nattrs, status
       integer start(2), edge(2), stride(2)
       integer datavar
       integer sfstart, sfselect, sfiscvar, sfginfo
       integer sfrdata, sfendacc, sfend

C      DFACC_RDONLY is defined in hdf.h. MAX_NC_NAME and MAX_VAR_DIMS
C      are defined in netcdf.h.
       integer*4 DFACC_RDONLY, MAX_NC_NAME, MAX_VAR_DIMS
       integer*4 X_LENGTH, Y_LENGTH
       parameter (DFACC_RDONLY = 1, MAX_NC_NAME = 256,
      +           MAX_VAR_DIMS = 32, X_LENGTH = 4, Y_LENGTH = 15)
       integer*2 array_data(X_LENGTH, Y_LENGTH)
       character name(MAX_NC_NAME)
       integer dims(MAX_VAR_DIMS)

C      Open the file and initiate the SD interface.
       sd_id = sfstart('Example3.hdf', DFACC_RDONLY)

C      Select the first (and in this case, only) data set in the file.
       sds_id = sfselect(sd_id, 0)

C      Confirm that the data set is not a coordinate variable.
       datavar = sfiscvar(sds_id)
       if (datavar .eq. 0) then

C         Verify the characteristics of the array.
          status = sfginfo(sds_id, name, rank, dims, nt, nattrs)

C         Define the location, pattern, and size of the data to read
C         from the data set.
          start(1) = 0
          start(2) = 0
          stride(1) = 1
          stride(2) = 1
          edge(1) = dims(1)
          edge(2) = dims(2)

C         Read the array data set.
```

```
               status = sfrdata(sds_id, start, stride, edge, array_data)
            endif

      C     Terminate access to the array data set.
            status = sfendacc(sds_id)

      C     Terminate access to the SD interface and close the file.
            status = sfend(sd_id)
            end
```

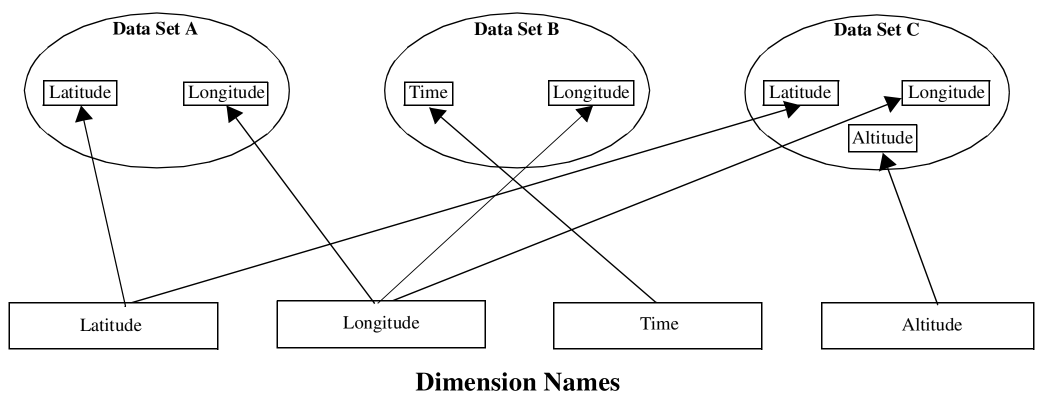
3.9.6 Dimension Scales for Multiple Data Sets

SD scientific data sets with one or more dimensions with the same name and size are considered to be related. Examples of related data sets are cross-sections from the same simulation, frames in an animation or images collected from the same apparatus. HDF attempts to preserve this relationship by unifying their dimension scales and attributes. To understand how related data sets are handled, it is necessary to understand how dimension records are created.

In the SD interface, dimension records are only created for dimensions of a unique name and size. To illustrate this, consider a case where there are three scientific data sets, each representing a unique variable, in an HDF file. (See Figure 3d.) The first two data sets have two dimensions each assigned to it and the third data set has three dimensions. There are a total of five dimensions in the file and the name mapping between the data sets and the dimensions are shown in the figure. Note that if, for example, the creation of a second dimension named "Altitude" is attempted and the size of the dimension is different from the existing dimension named "Altitude", an error condition will be generated.

As expected, assigning a dimension attribute to dimension 1 of either data set will create the required dimension scale and assign the appropriate attribute. However, because related data sets share dimension records, they also share dimension attributes. Therefore, it is impossible to assign an attribute to a dimension without assigning the same attribute to all dimensions of identical name and size, either within one data set or related data sets.

FIGURE 3d **Dimension Records and Attributes Shared Between Related Data Sets**



3.10 User-defined Attributes

User-defined attributes are attributes defined by the calling program that contains auxiliary information about a file, SDS array or dimension. This auxiliary information is sometimes called ***metadata***, because it is data about data. There are two ways to store metadata, as a user-defined attribute or predefined attribute.

They take the form `label=value`, where `label` is a character string containing `MAX_NC_NAME` or fewer characters and `value` contains one or more entries of the same data type as defined at the time the attribute is created. Attributes can be attached to three types of objects: files, data sets and dimensions. These are referred to, respectively, as ***file attributes*** , ***array attributes*** and ***dimension attributes***:

- ***File attributes*** are attributes that describe an entire file. They generally contain information pertinent to all data objects in the file and are sometimes referred to as ***global attributes***.
- ***Data set attributes*** are attributes that describe individual SDS arrays. Because their scope is limited to an individual SDS, data set attributes are sometimes referred to as ***local attributes***.
- ***Dimension attributes*** provide information applicable to an individual SDS dimension. It is possible to assign a unit to one dimension in a data set without assigning a unit to the remaining dimensions.

For each object, a separate ***attribute count*** is maintained that identifies the number of attributes associated with the object. The attribute count begins at zero and is increased by one for every new attribute assigned to an object. Each attribute associated with an object has a unique ***attribute index***, a value between 1 and the total number of attributes. The attribute index is used to retrieve an attribute's value or information about an attribute.

The data types permitted for attributes are the same as those allowed for SDS arrays. SDS arrays with general attributes of the same name can have different data types. For example, the attribute `valid_range` specifying the valid range of data value for an array of 16-bit integers might be of type 16-bit integer, whereas the attribute `valid_range` for a variable of 32-bit floats could be of type 32-bit floating-point integer.

Attribute names follow the same rules as dimension names. Providing meaningful names for attributes is important, however using standardized conventional names may be necessary if generic applications and utility programs are to be used. For example, every variable assigned a unit should have an attribute name of "units" defined. Furthermore, if an HDF file is to be used with software that recognizes "units" attributes, the values of the "units" attributes should be expressed in a conventional form as a character string that can be interpreted by that software.

The SD interface uses the same functions to access all attributes regardless of their object assignments. The difference between accessing a file, array or dimension attribute lies in the use of identifiers. File ids, SDS ids and dimension ids are used to respectively access file attributes, SDS attributes and dimension attributes.

3.10.1 Writing User-defined Attributes: SDsetattr

Attributes are not actually written out to a file until access to the object is terminated. Creating an attribute increases the attribute count by one for the given object. Writing an attribute involves the following steps:

1. Obtain the appropriate identifiers:
2. Create the attribute.
3. Terminate access by disposing of any existing identifiers:

To assign an attribute to a file, the calling program must contain a call to **SDsetattr**:

C: `status = SDsetattr(sd_id, attr_name, data_type, count, value);`

FORTRAN: `status = sfsnatt(sd_id, attr_name, data_type, count, value)`

In **SDsetattr** the argument `sd_id` is the identifier for the HDF object to be assigned the attribute and it can be a file id, SDS id or dimension id. The argument `attr_name` is an ASCII string containing the name of the attribute. It represents the label in the `label=value` equation and can be no more than `MAX_NC_NAME` characters. If this is set to the name of an existing attribute the value portion of the attribute will be overwritten. Do not use **SDsetattr** to assign a name to a dimension, use **SDsetdimname** instead.

The arguments, `data_type`, `count` and `value` describe the right side of the `label=value` equation. The `value` argument contain one or more values of the same data type. The `data_type` argument describes the data type for the attribute values and `count` defines the total number of values in the attribute.

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.

The parameters of **SDsetattr** are further described below. (See Table 3W on page 84.) Note that, because there are two Fortran-77 versions of **SDsetattr**, there are correspondingly two entries in the "Data Type" field of the *values* parameter.

EXAMPLE 19.

Setting Attribute Values

The following examples call **SDsetattr** with an SDS id as the first parameter, which assigns an SDS attribute to the selected data set. If a file id were passed instead, **SDsetattr** would assign a file attribute to the entire file.

C:

```
#include "hdf.h"
#include "mfhdf.h"

main( )
{

int32 sd_id, sds_id, dim_id, dim_index, status;
int32 num_values[2];

/* Open the file and get the identifier for the file. */
sd_id = SDstart("Example4.hdf", DFACC_RDWR);

/* Set an attribute that describes the file contents. */
status = SDsetattr(sd_id, "file_contents", DFNT_CHAR8, 16, \
                (VOIDP)"storm_track_data");

/* Get the identifier for the first data set. */
sds_id = SDselect(sd_id, 0);

/* Set an attribute the specifies a valid range of values. */
num_values[0] = 2;
num_values[1] = 10;
status = SDsetattr(sds_id, "valid_range", DFNT_INT32, 2, \
                (VOIDP)num_values);

/* Get the identifier for the first dimension. */
dim_id = SDgetdimid(sds_id, 0);

/* Set an attribute that specifies the dimension metric. */
status = SDsetattr(dim_id, "dim_metric", DFNT_CHAR8, 9,
                (VOIDP)"millibars");
```

```
/* Terminate access to the array */
status = SDendaccess(sds_id);

/* Terminate access to the SD interface and close the file */
status = SDend(sd_id);

}
```

FORTRAN:

```
      PROGRAM SET ATTRIBS

       integer*4 sd_id, sds_id, dim_id, status
       integer num_values(2)
       integer sfstart, sfsnatt, sfselect, sfdimid
       integer sfendacc, sfend
       integer DFACC_RDWR, DFNT_CHAR8, DFNT_INT32
       parameter (DFACC_RDWR = 3, DFNT_CHAR8 = 4, DFNT_INT32 = 24)

C      Open the file and get the identifier for the file.
       sd_id = sfstart('Example4.hdf', DFACC_RDWR)

C      Set an attribute that describes the file contents.
       status = sfsnatt(sd_id, 'file_contents', DFNT_CHAR8, 16,
     +                  'storm_track_data')
C      Get the identifier for the first data set.
       sds_id = sfselect(sd_id, 0)

C      Set an attribute the specifies a valid range of values.
       num_values(1) = 2
       num_values(2) = 10
       status = sfsnatt(sds_id, 'valid_range', DFNT_INT32, 2,
     +  num_values)

C      Get the identifier for the first dimension.
       dim_id = sfdimid(sds_id, 0)

C      Set an attribute that specifies the dimension metric.
       status = sfsnatt(dim_id, 'dim_metric', DFNT_CHAR8, 9,
     +  'millibars')

C      Terminate access to the array
       status = sfendacc(sds_id)

C      Terminate access to the SD interface and close the file
       status = sfend(sd_id)
       end
```

3.10.2 Querying User-defined Attributes: SDfindattr and SDattrinfo

Given a file, array or dimension id and an attribute name, **SDfindattr** will return a valid attribute index if the corresponding attribute exists. The attribute index can then be used to retrieve information about an attribute or its values. Given a file, array or dimension id and a valid attribute index **SDattrinfo** returns the name, data type and count for the corresponding attribute if it exists.

The syntax for **SDfindattr** and **SDattrinfo** is as follows:

C:

```
attr_index = SDfindattr(id, attr_name);
status = SDattrinfo(id, attr_index, attr_name, num_type, count);
```

```
FORTRAN: attr_index = sffattr(id, attr_name)
         status = sfgainfo(id, attr_index, attr, name, number_type,
                           count)
```

The parameters of **SDfindattr** and **SDattrinfo** are further described below. (See Table 3W.)

An attribute's index may also be determined by keeping track of the number and order of attributes as they are written or dumping the contents of a file using a dumping utility.

3.10.3 Reading User-defined Attributes: SDreadattr

SDreadattr reads the value or values of an attribute. The syntax for **SDreadattr** is as follows:

```
C:       status = SDreadattr(sds_id, attr_index, data);

FORTRAN: status = sfrattr(sd_id, attr_index, data)
```

SDreadattr takes a file, array, or dimension identifier and an attribute index specified in the `attr_index` parameter as input parameters and returns the attribute values in the buffer `data`. **SDreadaddr** will also read attributes and annotations created by the DFSD interface.

It's assumed that the buffer `data`, allocated to hold the attribute values, is large enough to hold the data. The size of the buffer must be at least `count*DFKNTsize(number_type)` bytes long. It is not possible to read a subset of values.

There are two Fortran-77 versions of this routine: **sfrnatt** and **sfrcatt**. The **sfrnatt** routine reads numeric attribute data and **sfrcatt** reads character attribute dataset.

The parameters of **SDreadattr** are further described in Table 3W. Note that, because there are two Fortran-77 versions of **SDreadattr**, there are correspondingly two entries in the "Data Type" field of the *data* parameter.

TABLE 3W **SDsetattr, SDfindattr, SDattrinfo and SDreadattr Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDsetattr (sfsnatt/ sfscatt)	file_id, sds_id or dim_id	int32	integer	File, array or dimension identifier.
	attr_name	char *	character* (*)	Attribute name.
	data_type	int32	integer	Data type of the attribute.
	count	int32	integer	Number of values in the attribute.
	values	VOIDP	<valid numeric data type>	Buffer for the data to be written.
SDfindattr (sffattr)	file_id, sds_id or dim_id	int32	integer	File, array or dimension identifier.
	attr_name	char *	character* (*)	Attribute name.

SDattrinfo (sfgainfo)	file_id, sds_id or dim_id	int32	integer	File, array or dimension identifier.
	attr_index	int32	integer	Index of the attribute to be read.
	attr_name	char *	character* (*)	Buffer for the name of the dimension attribute.
	data_type	int32 *	integer	Buffer for the data type of the values in the attribute.
	count	int32 *	integer	Buffer for the total number of values in the attribute.
SDreadattr (sfrnatt/ sfrcatt)	file_id, sds_id or dim_id	int32	integer	File, array or dimension identifier.
	attr_index	int32	integer	Index for the attribute to be read.
	data	VOIDP	<valid numeric data type>	Buffer for the attribute values.

EXAMPLE 20.

Retrieving Attribute Information

The attribute information stored in the "Example4.hdf" HDF file in Example 15 are read from the file in these examples.

C:

```
#include "hdf.h"
#include "mfhdf.h"

main( )
{

int32 sd_id, sds_id, status, *buffer1;
int32 attr_index, data_type, count;
char attr_name[MAX_NC_NAME];
int8 *buffer;

/* Open the file. */
sd_id = SDstart("Example4.hdf", DFACC_RDONLY);

/* Find the file attribute named "file_contents". */
attr_index = SDfindattr(sd_id, "file_contents");

/* Get information about the file attribute. */
status = SDattrinfo(sd_id, attr_index, attr_name, &data_type, &count);

/* Allocate a buffer to hold the attribute data. */
buffer = (int8 *)malloc(count * DFKNTsize(data_type));

/* Read the attribute data. */
status = SDreadattr(sd_id, attr_index, buffer);

/* Get the identifier for the first data set. */
sds_id = SDselect(sd_id, 0);

/* Find the data set attribute named "valid_range". */
attr_index = SDfindattr(sds_id, "valid_range");

/* Get information about the data set attribute. */
status = SDattrinfo(sds_id, attr_index, attr_name, &data_type, &count);

/* Allocate a buffer to hold the attribute data. */
buffer1 = (int32 *)malloc(count * DFKNTsize(data_type));
```

```
/* Read the attribute data. */
status = SDreadattr(sd_id, attr_index, buffer1);

/* Terminate access to the array */
status = SDendaccess(sds_id);

/* Terminate access to the SD interface and close the file */
status = SDend(sd_id);

}
```

FORTRAN:

```
     PROGRAM ATTRIB INFO

      integer*4 sd_id, sds_id, range_buffer(2)
      integer attr_index, data_type, count, status
      character attr_name *13
      character char_buffer *20
      integer sfstart, sffattr, sfgainfo, sfrattr, sfselect
      integer sfendacc, sfend

C     DFACC_RDWR is defined in hdf.h.
      integer DFACC_RDWR
      parameter (DFACC_RDWR = 3)

C     Open the file.
      sd_id = sfstart('Example4.hdf', DFACC_RDONLY)

C     Find the file attribute named 'file_contents'.
      attr_index = sffattr(sd_id, 'file_contents')

C     Get information about the file attribute.
      status = sfgainfo(sd_id, attr_index, attr_name, data_type,
     +  count)

C     Read the attribute data.
      status = sfrattr(sd_id, attr_index, char_buffer)

C     Get the identifier for the first data set.
      sds_id = sfselect(sd_id, 0)

C     Find the data set attribute named 'valid_range'.
      attr_index = sffattr(sds_id, 'valid_range')

C     Get information about the data set attribute.
      status = sfgainfo(sds_id, attr_index, attr_name, data_type,
     +  count)

C     Read the attribute data.
      status = sfrattr(sd_id, attr_index, range_buffer)

C     Terminate access to the array
      status = sfendacc(sds_id)

C     Terminate access to the SD interface and close the file
      status = sfend(sd_id)

      end
```

3.11 Predefined Attributes

Predefined attributes are attributes that use reserved labels and in some cases Predefined data types. Predefined attributes are categorized as follows:

- ***Labels*** can be thought of as independent variable names and dimension names and as such are used as primary search keys.
- ***Units*** are a means of declaring the units pertinent to a specific discipline. Unidata has developed a freely-available library of routines to convert between character string and binary forms of unit specifications and to perform useful operations on the binary forms. This library is used in some netCDF applications and is recommended for use with HDF applications. For more information, refer to the *netCDF User's Guide*.
- ***Formats*** describe the form numeric values will be printed and/or displayed. The recommended convention is to use standard Fortran-77 notation for describing the data format. For example, "F7.2" means to display seven digits with two digits to the right of the decimal point.
- ***Coordinate systems*** contain information that should be used when interpreting or displaying the data. For example, the text strings "cartesian", "polar" and "spherical" are recommended coordinate system descriptions.
- ***Ranges*** define the maximum and minimum values of a selected valid range. The range may cover the entire data set, values outside the data set or a subset of values within a data set. Because the HDF library does not check or update the range attribute as data is added or removed from the file, the calling program may assign any values deemed appropriate as long as they are of the same data type as the SDS array.
- A ***fill value*** is the value used to fill the areas between non-contiguous writes to SDS arrays. For more information about fill values, refer to Section 3.11.5 on page 91.
- ***Calibration*** stores scale and offset values used to create calibrated data in SDS arrays. When data are calibrated, they are typically reduced from floats, doubles or large integers into 8-bit or 16-bit integers and "packed" into an appropriately sized array. After the scale and offset values are applied, the packed array will return to its original form.

Predefined attributes are useful because they establish conventions that applications can depend on and because they are understood by the HDF library without users having to define them. Predefined attributes also insure backward compatibility with earlier versions of the HDF library. They can be assigned to two types of HDF objects: data sets and dimensions. (See Table 3X.)

TABLE 3X **Predefined Attribute List**

Object Type	Attribute	Reserved Label	Description
SDS Array or Dimension	Label	long_name	Name of the array.
	Unit	units	Units used for all dimensions and data.
	Format	format	Format for displaying dim scales and array values.
SDS Array Only	Coordinate System	cordsys	Coordinate system used to interpret the SDS array.
	Range	valid_range	Maximum and minimum values within a selected data range.
	Fill Value	_FillValue	Value used to fill empty locations in SDS array.
	Calibration	scale_factor	Value by which each array value is to be multiplied.
		scale_factor_err	Error introduced by scaling SDS array data.
		add_offset	Value to which each array value is to be added.
		add_offset_err	Error introduced by offsetting the SDS array data.
		calibrated_nt	Data type of the calibrated data.

The following naming conventions taken from the *netCDF User's Guide* are meant to promote consistency of information-sharing among generic applications. These naming conventions are not specifically required by the HDF library, but are highly recommended.

- **missing_value**: An attribute containing a value use to fill areas of an array not intended to contain valid data or a fill value. The scope of this attribute is local to the array. An example of this would be a region where information is unavailable, as in a geographical grid containing ocean data. The part of the grid where there is land might not have any data associated with it and in such a case the `missing_value` value could be supplied. The `missing_value` attribute is different from the `_FillValue` attribute in that fill values are intended to indicate data that was expected but did not appear, whereas missing values are used to indicate data that were never expected.
- **title**: A global file attribute containing a description of the contents of a file.
- **history**: A global file attribute containing the name of a program and the arguments used to derive the file. Well-behaved generic filters (programs that take HDF or netCDF files as input and produce HDF or netCDF files as output) would be expected to automatically append their name and the parameters with which they were invoked to the history attribute of an input file.

3.11.1 Accessing Predefined Attributes

The SD interface provides two methods for accessing predefined attributes. The first method uses the general attribute routines for user-defined attributes described in Section 3.11 on page 87 and the second employs routines specifically designed for each attribute. Although the general attribute routines work well and are recommended in most cases, the specialized attribute routines are sometimes easier to use, especially when reading or writing related predefined attributes. This is true for two reasons. First, because predefined attributes are guaranteed unique names, the attribute index is unnecessary. Second, attributes with several components may be read as a group. For example, using the SD routine designed to read the predefined calibration attribute returns all five components with a single call, rather than five separate calls.

There is one exception: unlike predefined array attributes, predefined dimension attributes should be read or written using the specialized attribute routines only.

The predefined attribute parameters are described in Table 3Y. Creating a predefined attribute with parameters different from these will produce unpredictable results when the attribute is read using the corresponding predefined-attribute routine.

TABLE 3Y **Predefined Attribute Parameter List**

Category	attr_name (Attribute type)	num_type (Data type)	count (Number of Values)	value (Attribute value)
Label	long_name	DFNT_CHAR8	string length	Pointer to string.
Unit	units	DFNT_CHAR8	string length	Pointer to string.
Format	format	DFNT_CHAR8	string length	Pointer to string.
Coordinate System	cordsys	DFNT_CHAR8	string length	Pointer to string.
Range	valid_range	<valid type>	2	Pointer to array.
Fill Value	_FillValue	<valid type>	1	Pointer to fill value.

Calibration	scale_factor	DFNT_FLOAT64	1	Pointer to scale.
	scale_factor_err	DFNT_FLOAT64	1	Pointer to scale error.
	add_offset	DFNT_FLOAT64	1	Pointer to offset.
	add_offset_err	DFNT_FLOAT64	1	Pointer to offset error.
	calibrated_nt	DFNT_INT32	1	Pointer to data type.

In addition to **SDreadattr**, **SDfindattr** and **SDattrinfo** are also valid general attribute routines to use when reading a predefined attribute. **SDattrinfo** is always useful for determining the size of an attribute whose value contains a string.

3.11.2 SDS String Attributes

Predefined string attributes for an SDS array include the ***label***, ***unit***, ***format*** and ***coordinate system*** of an SDS.

3.11.2.1 Writing String Attributes: SDsetdatastrs

The following function assigns label, unit, format and coordinate system string attributes to an SDS array:

C:
```
status = SDsetdatastrs(sds_id, label, unit, format, coordsys);
```

FORTRAN:
```
status = sfsdtstr(sds_id, label, unit, format, coordsys)
```

SDsetdatastrs assigns a predefined attribute to an SDS. The arguments for **SDsetdatastrs** are described in Table 3Z. To avoid creating one or more attributes, pass `NULL` as the appropriate argument.

3.11.2.2 Reading String Attributes: SDgetdatastrs

The following function reads the label, unit, format and coordinate system string attributes of an SDS:

C:
```
status = SDgetdatastrs(sds_id, label, unit, format, coordsys,
                       len);
```

FORTRAN:
```
status = sfgdtstr(sds_id, label, unit, format, coordsys, len)
```

SDgetdatastrs reads the predefined attributes of an SDS array. The arguments `label`, `unit`, `format` and `coordsys` are string buffers. If a particular attribute does not exist, the first character of the returned string will be `NULL`. Each string buffer is assumed to be at least `len` characters long, including the space to hold the `NULL` termination character. To avoid reading a particular attribute, pass `NULL` in the corresponding argument.

Keep in mind that the value returned by `label` is the value of the attribute named "long_name" and that the value returned by `coordsys` is the value of the attribute named "cordsys". The reasons for this are explained in User's Guide Appendix F.

The parameters of **SDgetdatastrs** are described in Table 3Z.

TABLE 3Z **SDsetdatastrs and SDgetdatastrs Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	

SDsetdatastrs (sfsdtstr)	sds_id	int32	integer	Data set identifier.
	label	char *	character* (*)	Label for the data.
	unit	char *	character* (*)	Definition of the units.
	format	char *	character* (*)	Description of the data format.
	coordsys	char *	character* (*)	Description of the coordinate system.
SDgetdatastrs (sfgdtstr)	sds_id	int32	integer	Data set identifier.
	label	char *	character* (*)	Buffer for the label.
	unit	char *	character* (*)	Buffer for the description of the units.
	format	char *	character* (*)	Buffer for the description of the data format.
	coordsys	char *	character* (*)	Buffer for the description of the coordinate system.
	len	integer	intn	Maximum length of the attributes.

3.11.3 Dimension String Attributes

Dimension string attributes include a ***label***, ***unit*** and ***format*** string which describe a dimension. They adhere to the same definitions as those of the label, unit and format strings for SDS attributes.

3.11.3.1 Writing a Dimension String Attribute: SDsetdimstrs

C: `status = SDsetdimstrs(dim_id, label, unit, format);`

FORTRAN: `status = sfsdmstr(dim_id, label, unit, format)`

SDsetdimstrs assigns label, unit and format attributes to an SDS dimension. The `dim_id` argument is the dimension identifier returned by the call to **SDgetdimid**. It identifies the dimension to which the attribute will be assigned. The remaining arguments are described below. (See Table 3AA on page 90.)

3.11.3.2 Reading a Dimension String Attribute: SDgetdimstrs

C: `status = SDgetdimstrs(dim_id, label, unit, format, len);`

FORTRAN: `status = sfgdmstr(dim_id, label, unit, format, len)`

SDgetdimstrs reads the attributes specified by the `label`, `unit` and `format` parameters to an SDS dimension. The arguments `label`, `unit`, and `format` are buffers to hold the label, unit and format strings as defined in their respective attributes. If a particular attribute does not exist, the first character of the returned string will be `NULL`. Each buffer is assumed to be at least `len` characters long including the space to hold the `NULL` termination character. To avoid reading a particular attribute, pass `NULL` as the appropriate argument.

The parameters of **SDgetdimstrs** are described in Table 3AA.

TABLE 3AA **SDsetdimstrs and SDgetdimstrs Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type C	Data Type Fortran-77	Description
SDsetdimstrs (sfsdmstr)	dim_id	int32	integer	Dimension identifier.
	label	char *	character* (*)	Label describing the specified dimension.
	unit	char *	character* (*)	Units to be used with the specified dimension.
	format	char *	character* (*)	Format to use when displaying the scale values.

SDgetdimstrs (sfgdmstr)	dim_id	int32	integer	Dimension identifier.
	label	char *	character* (*)	Buffer for the dimension label.
	unit	char *	character* (*)	Buffer for the dimension unit.
	format	char *	character* (*)	Buffer for the dimension format.
	len	intn	integer	Maximum length of the string attributes.

3.11.4 Range Attributes

The attribute ***range*** contains user-defined maximum and minimum values in a selected range. As the HDF library does not check or update the range attribute as data is added or removed from the file, the calling program may assign any values deemed appropriate. Also, because the maximum and minimum values are supposed to relate to the data set, it is assumed that they are of the same data type as the data.

3.11.4.1 Writing a Range Attribute: SDsetrange

SDsetrange assigns the range attribute to an SDS:

C: `status = SDsetrange(sds_id, max, min);`

FORTRAN: `status = sfsrange(sds_id, max, min)`

The parameters of **SDsetrange** are described below. (See Table 3AB on page 91.)

3.11.4.2 Reading a Range Attribute: SDgetrange

SDgetrange reads the maximum and minimum valid values of an SDS array as specified by a **SDsetrange** call or its equivalent:

C: `status = SDgetrange(sds_id, max, min);`

FORTRAN: `status = sfgrange(sds_id, max, min)`

The arguments `max` and `min` are buffers the maximum and minimum values will be read into. The arguments in the C version of **SDgetrange** are pointers rather than simple variables, whereas in the Fortran-77 version they are variables of the same data type as the data set. The parameters of **SDgetrange** are described in the following table.

TABLE 3AB **SDsetrange and SDgetrange Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDsetrange (sfsrange)	sds_id	int32	integer	Data set identifier.
	max	VOIDP	<valid numeric data type>	Maximum value of the range.
	min	VOIDP	<valid numeric data type>	Minimum value of the range.
SDgetrange (sfgrange)	sds_id	int32	integer	Data set identifier.
	max	VOIDP	<valid numeric data type>	Buffer for the maximum value.
	min	VOIDP	<valid numeric data type>	Buffer for the minimum value.

3.11.5 Fill Values

A ***fill value*** is the value used to fill the spaces between non-contiguous writes to SDS arrays. If a fill value is set before writing data to an SDS, the entire array is initialized to the specified fill value. By default, any location not subsequently overwritten by SDS data will contain the fill value.

A fill value must be of the same data type as the array to which it's written. To avoid conversion errors, use data-specific fill values instead of special architecture-specific values, such as infinity and ***Not-a-Number*** or ***NaN***.

Fill values can also be defined for all SDSs within a file. This is determined by setting a ***fill mode***, which can be done by calling the **SDsetfillmode** routine described below.

3.11.5.1 Writing a Fill Value Attribute: SDsetfillvalue

SDsetfillvalue assigns a new value to the fill value attribute for an SDS array:

C: `status = SDsetfillvalue(sds_id, fill_val);`

FORTRAN: `status = sfsfill(sds_id, fill_val)`

The argument `fill_val` is the new fill value. It is recommended that you set the fill value before writing data to an SDS array, as calling **SDsetfillvalue** after data is written to an SDS array only changes the fill value attribute - it does not update the existing fill values.

There are two Fortran-77 versions of this routine: **sfsfill** and **sfscfill**. The **sfsfill** routine writes numeric fill value data and **sfscfill** writes character fill value data.

The parameters of **SDsetfillvalue** are described below. (See Table 3AC on page 93.) Note that, because there are two Fortran-77 versions of **SDsetfillvalue**, there are correspondingly two entries in the "Data Type" field of the *fill_val* parameter.

3.11.5.2 Reading a Fill Value Attribute: SDgetfillvalue

SDgetfillvalue reads in the fill value of an SDS array as specified by a **SDsetfillvalue** call or its equivalent:

C: `status = SDgetfillvalue(sds_id, fill_val);`

FORTRAN: `status = sfgfill(sds_id, fill_val)`

The argument `fill_val` is the space allocated to store the fill value.

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.

The parameters of **SDgetfillvalue** are described in the following table. Note that, because there are two Fortran-77 versions of **SDgetfillvalue**, there are correspondingly two entries in the "Data Type" field of the `fill_val` parameter.

3.11.5.3 Setting the Fill Mode for all SDSs in the Specified File: SDsetfillmode

Writing fill values to an SDS can involve more I/O overhead than is necessary. This is because, whenever a fill value is set for an SDS, two write operations are generally needed - one to write the fill value and one to write the actual dataset data. It is "generally needed" because, whenever all of the data is written to the dataset in one write operation the additional write operation to add the fill values is not performed, as it isn't necessary. For datasets containing contiguous data, preventing the HDF library from performing these fill value write operations can result in a substantial performance increase.

However, it can be tedious to unset the fill value for all SDSs in a file, particularly when the file contains a large number of them. This can be done with one call to **SDsetfillmode**, which must occur before calls to **SDsetfillvalue**.

The syntax of **SDsetfillmode** is as follows:

C:
```
status = SDsetfillmode(file_id, fill_mode);
```

FORTRAN:
```
status = sfsflmd(file_id, fill_mode)
```

The argument `file_id` is the identifier of the file the fill mode will be applied to. The `fill_mode` argument is the fill mode to be applied - it can be set to either `SD_FILL` or `SD_NOFILL`. `SD_FILL` specifies that fill values will be written to all SDSs in the specified file by default. If **SDsetfillmode** is never called before **SDsetfillvalue**, this is the default fill mode. `SD_NOFILL` specifies that, by default, fill values will not be written to all SDSs in the specified file. This can be overridden for specific SDSs by subsequent calls to **SDsetfillvalue**.

Note that, whenever a file has been newly opened, or has been closed and then re-opened, the default `SD_FILL` fill mode will be in effect until it is changed by a call to **SDsetfillmode**.

The parameters of **SDsetfillmode** are described in the table below.

TABLE 3AC **SDsetfillvalue, SDgetfillvalue and SDsetfillmode Parameter List,**

Function Call (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDsetfillvalue (sfsfill/ sfscfill)	sds_id	int32	integer	Data set identifier.
	fill_val	VOIDP	<valid numeric data type>	Pointer to the fill value to be set.
SDgetfillvalue (sfgfill/ sfgcfill)	sds_id	int32	integer	Data set identifier
	fill_val	VOIDP	<valid numeric data type>	Buffer for the fill value.
SDsetfillmode (sfsflmd)	file_id	int32	integer	File identifier.
	fill_mode	intn	integer	Fill mode to be set.

3.11.6 Calibration Attributes

The ***calibration*** attribute stores scale and offset values to describe calibrated data in SDS arrays. When data are calibrated using a scale and an offset, the values in an array can be represented using a smaller data type than the original. For instance, an array containing data of type `float` could be stored as an array containing data of type 8- or 16-bit integer. Both the `scale_factor` and `add_offset` attributes should be of the type `float64`.

3.11.6.1 Writing Calibrated Data: SDsetcal

SDsetcal adds the scale factor, offset, scale factor error, offset error and the data type of the uncalibrated data to the specified data set:

C:
```
status = SDsetcal(sds_id, cal, cal_error, offset, off_err,
                  num_type);
```

FORTRAN:
```
status = sfscal(sds_id, cal, cal_error, offset, off_err,
                num_type)
```

SDsetcal must be called to calibrate the data before the data is written.

SDsetcal has six arguments; `sds_id`, `cal`, `cal_error`, `offset`, `off_err` and `num_type`. The argument `cal` represents a single value that when multiplied against every value in the calibrated array reproduces the original data (assuming an `offset` of 0). The argument `offset` represents a single value that when subtracted from every value in the calibrated array reproduces the original data (assuming a `scale` of 1). The `cal` and `offset` values relate to the original data according to the following equation:

```
original_data = cal * (calibrated_data - offset)
```

In addition to `cal` and `offset`, **SDsetcal** also includes both a scale and offset error. The argument `cal_err` contains the potential error of the calibrated data due to scaling; `off_err` contains the potential error for the calibrated data due to the offset.

The parameters of **SDsetcal** are described below. (See Table 3AD on page 94.)

3.11.6.2 Reading Calibrated Data: SDgetcal

SDgetcal reads calibration attributes for an SDS array as written by a **SDsetcal** call or its equivalent:

C:
```
status = SDgetcal(sds_id, cal, cal_error, off, off_err,
                  num_type);
```

FORTRAN:
```
status = sfgcal(sds_id, cal, cal_error, off, off_err, num_type)
```

Because the HDF library does not actually apply calibration information to the data, **SDgetcal** can be called anytime before or after the data is read. If a calibration record does not exist, **SDgetcal** returns `FALSE`. **SDgetcal** takes six arguments; `sds_id`, `cal`, `cal_error`, `offset`, `off_err` and `num_type`. These parameters are described in the following table.

TABLE 3AD

SDsetcal and SDgetcal Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
SDsetcal (sfscal)	sds_id	int32	integer	Data set identifier.
	cal	float64	real*8	Calibration factor.
	cal_error	float64	real*8	Calibration error.
	off	float64	real*8	Uncalibrated offset.
	off_err	float64	real*8	Uncalibrated offset error.
	num_type	int32	integer	Data type of uncalibrated data.
SDgetcal (sfgcal)	sds_id	int32	integer	Data set identifier.
	cal	float64 *	real*8	Pointer to the calibration factor.
	cal_error	float64 *	real*8	Pointer to the calibration error.
	off	float64 *	real*8	Pointer to the uncalibrated offset.
	off_err	float64 *	real*8	Pointer to the uncalibrated offset error.
	num_type	int32 *	integer	Pointer to the data type of uncalibrated data.

EXAMPLE 21.

Calibrating Data

Suppose the values in the calibrated array `cal_val` are the following integers:

```
cal_val[6] = {2, 4, 5, 11, 26, 81}
```

By applying the calibration equation `orig = cal * (cal_val - offset)` with `cal = 0.50` and `offset = -2000.0`, the calibrated array `cal_val[]` returns to its original floating-point form:

```
original_val[6] = {1001.0, 1002.0, 1002.5, 1005.5, 1013.0, 1040.5}
```

3.12 netCDF

HDF supports the netCDF data model and interface developed at the Unidata Program Center (UPC). Like HDF, netCDF is an interface to a library of data access programs that store and retrieve data. The file format developed at the UPC to support netCDF uses XDR (eXternal Data Representation) a non-proprietary external data representation developed by Sun Microsystems for describing and encoding data . Full documentation on netCDF and the Unidata netCDF API is available at `http://www.unidata.ucar.edu/packages/netcdf/`.

The netCDF data model is interchangeable with the SDS data model in so far as it is possible to use the netCDF calling interface to place an SDS into an HDF file and conversely the SDS interface will read from an XDR-based netCDF file. Because the netCDF API has not changed and netCDF files stored in XDR format are readable, existing netCDF programs and data are still usable, although programs will need to be relinked to the new library. However, there are important conceptual differences between the HDF and the netCDF data model that must be understood to effectively use HDF for the purpose of working with netCDF data objects and to understand enhancements to the API that will be included in the future to make the two APIs much more similar.

In the HDF model, when an n-dimensional SDS is created by **SDcreate**, data objects are also created that provide information about the individual dimensions - one for each dimension. Each SDS contains within its internal structure the array data as well as pointers to these dimensional data objects. Each dimensional data object is stored in a structure that is in the HDF file, but separate from the SDS array.

If more than one SDS have the same dimension sizes, they may share dimensions by pointing to the same dimensional data objects. This can be done in application programs by calling **SDsetdimname** to assign the same dimension name to all dimensions that are shared by several SDS objects. For example, suppose you make the following sequence of calls for every SDS in a file:

```
dim_id = SDgetdimid(sds_id, 0);
ret = SDsetdimname(dim_id, "Lat");
dim_id = SDgetdimid(sds_id, 1);
ret = SDsetdimname(dim_id, "Long");
```

This will cause every SDS to refer to the dimensional data object named "Lat" as its first dimension and to the dimensional data object named "Long" as its second dimension.

This same result is obtained differently in netCDF. Note that a netCDF "variable" is roughly the same as an HDF SDS. The netCDF API requires application programs to define all dimensions, using `ncdimdef`, before defining variables. Those defined dimensions are then used to define variables in `ncvardef`. Each dimension is defined by a name and a size. All variables using the same dimension will have the same dimension name and dimension size.

Although the HDF SDS API will read from and write to *existing* XDR-based netCDF files, HDF cannot be used to *create* XDR-based netCDF files.

There is currently no support for the mixing of HDF objects that are not SDSs and netCDF objects. For example, a raster image can exist in the same HDF file as a netCDF object, but you must use one of the HDF raster image interfaces to read the image and the HDF SD or netCDF interface to read the netCDF object. The other HDF interfaces are currently being modified to allow multifile access. Closer integration with the netCDF interface will probably be delayed until the end of that project.

3.12.1 HDF Interface vs. netCDF Interface

Existing netCDF applications can be used to read HDF files and existing HDF applications can be used to read XDR-based netCDF files. To read an HDF file using a netCDF application, the application must be recompiled using the HDF library. For example, recompiling the netCDF utility `ncdump` with HDF creates a utility that can dump scientific data sets from both HDF and XDR-based files. To read an XDR-based file using an HDF application, the application must be relinked to the HDF library.

The current version of HDF contains three APIs that support essentially the same data model:

- The multifile SD interface.
- The netCDF or NC interface.
- The single-file DFSD interface.
- The multifile GR interface.

The first three interfaces can create, read and write SDSs in HDF files. Both the SD and NC interfaces can read from and write to XDR-based netCDF files, but they cannot create them. This interoperability means that a single program may contain both SD and NC function calls and thus transparently read and write scientific data sets to HDF or XDR-based files.

The SD interface is the only HDF interface capable of accessing the XDR-based netCDF file format. The DFSD interface cannot access XDR-based files and can only access SDS arrays, dimension scales and predefined attributes. A summary of file interoperability among the three interfaces is provided in the following table.

TABLE 3AE **Summary of HDF and XDR File Compatibility for the HDF and netCDF APIs**

	Files Created by DFSD Interface	Files Created by SD Interface	Files Written by NC Interface	
	HDF	HDF	NCSA HDF Library	Unidata netCDF Library
Accessed by DFSD	Yes	Yes	Yes	No
Accessed by SD	Yes	Yes	Yes	Yes
Accessed by NC	Yes	Yes	Yes	Yes

A summary of NC function calls and their SD equivalents is presented in the following table.

TABLE 3AF

NC Interface Routine Calls and Their SD Equivalents

Purpose	Routine Name		SD Equivalent	Description
	C	Fortran-77		
Operations	nccreate	NCCRE	SDstart	Creates a file.
	ncopen	NCOPN	SDstart	Opens a file.
	ncredef	NCREDF	Not Applicable	Sets open file into define mode.
	ncendef	NCENDF	Not Applicable	Leaves define mode.
	ncclose	NCCLOS	SDend	Closes an open file.
	ncinquire	NCINQ	SDfileinfo	Inquires about an open file.
	ncsync	NCSNC	Not Applicable	Synchronizes a file to disk.
	ncabort	NCABOR	Not Applicable	Backs out of recent definitions.
	ncsetfill	NCSFIL	Not Implemented	Sets fill mode for writes.
Dimensions	ncdimdef	NCDDEF	SDsetdimname	Creates a dimension.
	ncdimid	NCDID	SDgetdimid	Returns a dimension identifier from its name.
	ncdiminq	NCDINQ	SDdiminfo	Inquires about a dimension.
	ncdimrename	NCDREN	Not Implemented	Renames a dimension.
Variables	ncvardef	NCVDEF	SDcreate	Creates a variable.
	ncvarid	NCVID	SDnametoindex and SDselect	Returns a variable identifier from its name.
	ncvarinq	NCVINQ	SDgetinfo	Returns information about a variable.
	ncvarput1	NCVPT1	Not Implemented	Writes a single data value.
	ncvarget1	NCVGT1	Not Implemented	Reads a single data value.
	ncvarput	NCVPT	SDwritedata	Writes a hyperslab of values.
	ncvarget	NCVGT/NCVGTC	SDreaddata	Reads a hyperslab of values.
	ncvarrename	NCVREN	Not Implemented	Renames a variable.
	nctypelen	NCTLEN	DFKNTsize	Returns the number of bytes for a data type.
Attributes	ncattput	NCAPT/NCAPTC	SDsetattr	Creates an attribute.
	ncattinq	NCAINQ	SDattrinfo	Returns information about an attribute.
	ncattcopy	NCACPY	Not Implemented	Copies attribute from one file to another.
	ncattget	NCAGT/NCAGTC	SDreadattr	Returns attributes values.
	ncattname	NCANAM	SDattrinfo	Returns name of attribute from its number.
	ncattrename	NCAREN	Not Implemented	Renames an attribute.
	ncattdel	NCADEL	Not Implemented	Deletes an attribute.

Chapter 4

Vdatas (VS API)

4.1 Chapter Overview

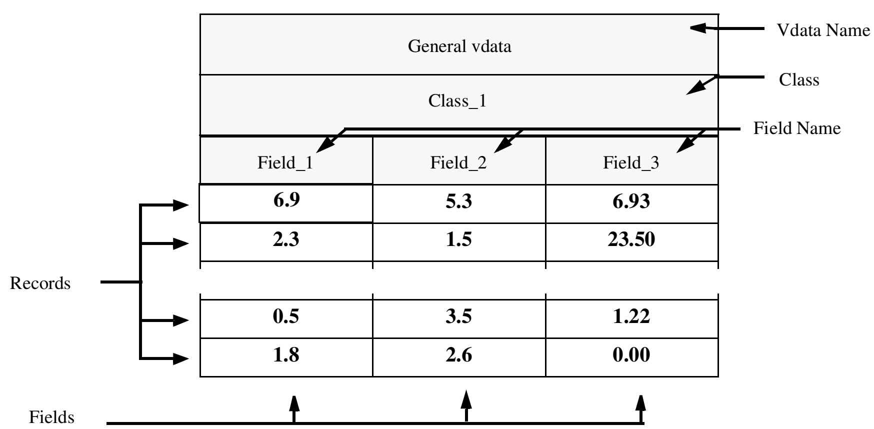
This chapter describes the vdata data model, the vdata API (also called the VS interface) and the vdata programming model. The last section of this chapter introduces command-line utilities that operate on vdata objects.

4.2 The Vdata Model

The vdata object is a collection of records whose values are stored in fixed-length fields. The HDF ***Vdata model*** provides a framework for storing customized tables, or ***vdatas***, in HDF files. The term "vdata" is an abbreviation of "vertex data" which alludes to the fact that, when the object was first implemented in HDF, it was designed specifically for the purpose of storing the vertex and edge information of polygon sets. The vdata design has since been generalized to apply to a broader variety of applications.

Vdatas are uniquely identified by a ***name***, a ***class*** and a series of individual ***field names*** . (See Figure 4a.)

FIGURE 4a **Vdata Table Structure**



A ***vdata name*** is a label typically assigned to describe the contents of a vdata. It often serves as a search key for files containing multiple vdatas. A ***vdata class*** further distinguishes a particular vdata by identifying the purpose or use of its data. Finally, ***vdata field names*** are labels assigned to each field.

4.2.1 Records and Fields

A vdata is a collection of ***records***, which itself is comprised of one or more fixed-length ***fields***. A vdata is like a table in which each row has the same structure and each column contains data of the same data type. (See Figure 4b.) Vdata records and fields are identified by an index. The record and field indexes are zero-based and separately incremented by one for each additional field and record in the vdata object.

Every field in a vdata is assigned a data type when the vdata is created. The data type of a field may be any basic HDF data type: character, 8-bit, 16-bit and 32-bit signed and unsigned integers, and 32-bit and 64-bit floating point numbers. The maximum length of a vdata record is 32,767 bytes.

The Vdata model allows multiple entries per field. The number of entries or ***components*** in a field is more commonly called the ***order*** of the field.

The organizational structure of a vdata is often determined by the data types of its data set or sets. For example, given a data set describing the location ("X, Y") and temperature ("Temp") of points in a plane, there are several ways to organize the data. (See Figure 4b.) If the "X", "Y" and "Temp" values are of the same data type, they could be stored as three single-component fields, as two-component "X, Y" field and a Temp field or as a three-component "X, Y, Temp" field. Generally, the "X,Y" data is kept in a single field, but HDF places no restrictions on the organization of field data and there are no significant HDF performance issues involved in choosing one organizational regime over another.

FIGURE 4b **Three Different Vdata Structures for Data of the Same Number Type**

Simulation Data 1		
2D_Temperature_Grid		
X	Y	Temp
2.30	**1.50**	**23.50**
3.40	**5.70**	**8.03**
0.50	**3.50**	**1.22**
1.80	**2.60**	**0.00**

3 Single-Component Fields

Simulation Data 1	
2D_Temperature_Grid	
X, Y	Temp
2.30, 1.50	**23.50**
3.40, 5.70	**8.03**
0.50, 3.50	**1.22**
1.80, 2.60	**0.00**

1 Multi-Component Field
1 Single-Component Field

Simulation Data 1
2D_Temperature_Grid
X, Y, Temp
2.30, 1.50, 23.50
3.40, 5.70, 8.03
0.50, 3.50, 1.22
1.80, 2.60, 0.00

1 Multi-Component Field

4.2.2 Conventions

To facilitate the sharing of data in an HDF data file among different users and organizations, it is a good idea to establish conventions about vdata attributes such as vdata names, classes, field and record structures and field names. A typical HDF convention describes a configuration of vdata and vgroup objects designed to store polygonal data. These vdatas and vgroups collectively form a structure called a ***vset***. Chapter 5, titled *Vgroups (V API)*, contains additional information on vsets.

4.3 The Vdata API

The VS API consists of routines that are used to store and retrieve information about vdatas and their contents. The V interface, which is described in Chapter 5, titled *Vgroups (V API)*, can also be used to perform certain operations on vdatas.

4.3.1 Vdata Library Routines

Vdata routines begin with the prefixes "VS", "VF", "VSQ" and "VH". Routines in the V interface are prefaced by a capital "V". V routines perform most general vdata operations, VF routines query information about vdata fields and VSQ routines query information about specific vdatas. VH routines are high-level procedures designed to write to single-field vdatas.

Vdata routines let you define, organize and manipulate vdatas, and are categorized as follows:

- ***Access routines*** attach, or allow access, to vdatas. Data transfer can only occur after a vdata has been accessed. These routines also detach from, or properly terminate access to, vdatas when data transfer is completed.
- ***Read and write routines*** read and write the contents of a vdata.
- ***File inquiry routines*** provide information about how vdatas are stored in a file. They are useful for locating vdatas in a file.
- ***Vdata inquiry routines*** provide specific information about a given vdata, including the vdata's name, class, number of fields, number of records, tag and reference number pairs, interlace mode and size.
- ***Field inquiry routines*** provide specific information about the fields in a given vdata, including the field's size, name, order, type and number of fields in the vdata.

TABLE 4A **Vdata Interface Routines**

Category	Routine Name		Description
	C	Fortran-77	
Access	Vend	vfend	Closes the vdata interface.
	Vstart	vfstart	Initializes the vdata interface.
	VSattach	vsfatch	Establishes access to a specified vdata.
	VSdetach	vsfdtch	Terminates access to a specified vdata.
Read and Write	VHstoredata	vhfsd/vhfscd	Writes data to a simple, single-component vdata.
	VHstoredatam	vhfsdm/vhfscdm	Writes a vdata with one multi-component field.
	VSfdefine	vsffdef	Defines a new vdata field.
	VSsetclass	vsfscls	Assigns a class to a vdata.
	VSsetfields	vsfsfld	Specifies the vdata fields to be written to.
	VSsetinterlace	vsfsint	Sets the interlace mode for a vdata.
	VSsetname	vsfsnam	Assigns a name to a vdata.
	VSwrite	vsfwrit	Writes records to a vdata.
	VSread	vsfread	Reads from a vdata.
	VSseek	vsfseek	Seeks to a specified record in a vdata.
	VSsetattr	vsfsnat/vsfscat	Sets the attribute of a vdata field or vdata.
File Inquiry	VSfind	vsffnd	Searches for a given vdata name the opened HDF file.
	VSgetid	vsfgid	Returns the identifier of the next vdata in the file.
	VSlone	vsflone	Returns the vdatas that are not linked into vgroups.

Vdata Inquiry	VSfexist	vsfex	Tests for the existence of fields in the specified vdata.
	VSinquire	vsfinq	Returns information about the specified vdata.
	VSelts	vsfelts	Returns the number of records in the specified vdata.
	VSgetclass	vsfcls	Returns the class name of the specified vdata.
	VSgetfields	vsfgfld	Returns all field names within the specified vdata.
	VSgetinterlace	vsfgint	Retrieves the interlace mode of the specified vdata.
	VSgetname	vsfgnam	Retrieves the name of the specified vdata.
	VSsizeof	vsfsiz	Returns the field sizes of the specified vdata.
	VSQueryclass	None	Returns the class of the specified vdata.
	VSQueryfields	vsgfld	Returns the field names of the specified vdata.
	VSQueryname	vsfgname	Returns the name of the specified vdata.
	VSQueryref	None	Retrieves the reference number of the specified vdata.
	VSQuerytag	None	Retrieves the tag of the specified vdata.
	VSQuerycount	vsfelts	Returns the number of records in the specified vdata.
	VSQueryinterlace	vsfgint	Returns the interlace mode of the specified vdata.
	VSQueryvsize	vsfsiz	Retrieves the local size in bytes of the specified vdata record.
	VSfindex	vsffidx	Queries the index of a vdata field given the field name.
	VSsetattr	vsfsnat/vsfscat	Sets the attribute of a vdata field or vdata.
	VSnattrs	vsfnats	Queries the total number of vdata attributes.
	VSfnattrs	vsffnas	Queries the number of attributes of a vdata or vdata field.
	VSfindattr	vsffdat	Retrieves the index of an attribute given the attribute name.
	VSattrinfo	vsfainf	Queries information on a given attribute.
	VSgetattr	vsfgnat/vsfgcat	Queries the values of a given attribute.
	VSisattr	vsfisat	Determines if the given vdata is an attribute.
Field Inquiry	VFfieldesize	None	Retrieve the field size (as stored in a file) of a specified field.
	VFfieldisize	None	Retrieve the field size (as stored in memory) of a specified field.
	VFfieldname	None	Retrieves the name of the specified field in the given vdata.
	VFfieldorder	None	Retrieves the order of the specified field in the given vdata.
	VFfieldtype	None	Retrieves the data type for the specified field in the given vdata.
	VFnfields	None	Retrieves the total number of fields in the specified vdata.

4.3.2 Vdata Identifiers in the Vdata API

The VS interface identifies vdatas in several ways. Before a vdata is opened or *attached*, it is identified by its name, class and reference number. When a vdata is attached it is assigned a vdata identifier, or ***vdata id***, which is used by the vdata interface routines in accessing vdatas.

The name and class are assigned by the calling program and may be changed throughout the scope of the vdata object.

4.3.3 Programming Model for the Vdata Interface

The programming model for accessing vdatas is as follows:

1. Open a file.
2. Initialize the VS interface.
3. Open a vdata by obtaining a vdata id from a vdata reference number.
4. Perform the desired operations on the vdata.
5. Dispose of the vdata id.

6. Terminate access to the VS interface.
7. Close the file.

To access a vdata, the calling program must contain the following calls:

C:

```
file_id = Hopen(filename, file_access_mode, n_dds);
status = Vstart(file_id);
vdata_id = VSattach(file_id, vdata_ref, vdata_access_mode);

<Optional operations>

status = VSdetach(vdata_id);
status = Vend(file_id);
status = Hclose(file_id);
```

FORTRAN:

```
file_id = hopen(filename, file_access_mode, n_dds)
status = vfstart(file_id)
vdata_id = vsfatch(file_id, vdata_ref, vdata_access_mode)

<Optional operations>

status = vsfdtch(vdata_id)
status = vfend(file_id)
status = hclose(file_id)
```

4.3.4 Accessing Files and Vdatas: Hopen, Vstart and VSattach

The general routine **Hopen** opens an HDF file and obtains a file identifier. Details on the use of **Hopen** are provided in Chapter 2, titled *HDF Fundamentals*.

To perform operations on a vdata, the calling program must call **Vstart** for every file to be accessed and **VSattach** for every vdata to be accessed. **Vstart** initializes the internal vdata structures used by the VS API. **Vstart** has as its only argument the file identifier returned by **Hopen**.

VSattach accesses an individual vdata and must be called before all read and write operations. It has three arguments: `file_id`, `vdata_ref` and the `file_access_mode`. The `file_id` argument is the file identifier returned by **Hopen** and `vdata_ref` is the reference number that identifies to vdata to be accessed. Specifying a nonexistent reference number will return an error, specifying a value of `-1` will create a new vdata and specifying an existing reference number will result in the corresponding vdata being accessed.

The `access_mode` argument specifies the access mode ("`r`" for read-only access or "`w`" for write-only) required for subsequent operations on the specified vdata. Although several HDF user programs may simultaneously read from one vdata object, only one write access is allowed at a time. The "`r`" read-only access mode may only be used with existing vdatas, but the "`w`" access mode is valid with both new vdatas (`vdata_ref = -1`) and existing vdatas.

The parameters for **Vstart** and **VSattach** are further defined below. (See Table 4B.)

4.3.5 Terminating Access to Vdatas and Files: VSdetach, Vend and Hclose

VSdetach terminates access by updating pertinent information and freeing all memory associated with the vdata. Once access to the vdata is terminated, its identifier is invalid and any attempt to access it will result in an error condition.

Vend releases all internal data structures allocated by **Vstart**. It takes one argument the file identifier returned by **Hopen**. **Vend** must be called once per file and only after access to all vdata objects are terminated. Attempts to call VS interface routines after calling **Vend** will result in an error condition.

Successfully terminating access to a vdata requires one **VSdetach** call for each call to **VSattach** and one **Vend** call for each call to **Vstart**. All **VSdetach** calls must be made prior to making the first call to **Vend**.

Hclose terminates access to a file only if no vdata ids remain allocated. It is described in Chapter 2, titled *HDF Fundamentals*.

The parameters for **VSdetach** and **Vend** are further defined in the following table.

TABLE 4B

Vstart, VSattach, VSdetach and Vend Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
Vstart (vfstart)	file_id	int32	integer	File identifier.
VSattach (vsfatch)	vdata_id	int32	integer	Vdata identifier.
	vdata_ref	int32	integer	Reference number of the vdata.
	file_access_mode	char *	character*1	Vdata access mode.
VSdetach (vsfdtch)	vdata_id	int32	integer	Vdata identifier.
Vend (vfend)	file_id	int32	integer	File identifier.

EXAMPLE 1.

Accessing a Vdata in an HDF File

These examples show how to write vdata information. It returns information about the specified vdata. The **VSinquire** routine is described in Section 4.9 on page 135.

Note that the “Dummy_HDF_File.hdf” HDF file represents a previously-created file.

C:

```
#include “hdf.h”

main( )
{

int32 file_id, vdata_id, status;
int32n_records, interlace, vdata_size;
charfields[30], vdata_name[30];

/* Open the HDF file. */
file_id = Hopen(“Dummy_HDF_File.hdf”, DFACC_RDONLY, 0);

/* Initialize the VS interface. */
status = Vstart(file_id);

/* Get information about the vdata. */
status = VSinquire(vdata_id, &n_records, &interlace, fields, \
                    &vdata_size, vdata_name);

/* Terminate access to the vdata. */
```

```
status = VSdetach(vdata_id);

/* Terminate access to the VS interface. */
status = Vend(file_id);

/* Close the HDF file. */
status = Hclose(file_id);

}
```

FORTRAN:

```
      PROGRAM VDATA ACCESS

      integer*4 file_id, vdata_id, n_records, interlace, vdata_size
      integer status
      character fields*30, vdata_name*30
      integer hopen, vfatch, vsfinq, vfdtch, hclose

C     DFACC_RDONLY is defined in 'hdf.inc'.
      integer DFACC_RDONLY
      parameter (DFACC_RDONLY = 1)

C     Open an HDF file with read-only access.
      file_id = hopen('Dummy_HDF_File.hdf', DFACC_RDONLY, 0)

C     Initialize the Vset interface.
      status = vfstart(file_id)

C     Attach to the first Vdata.
      vdata_id = vfatch(file_id, -1, 'w')

C     Get information about the Vdata.
      status = vsfinq(vdata_id, n_records, interlace, fields,
     +                            vdata_size, vdata_name)

C     Terminate access to the Vdata.
      status = vfdtch(vdata_id)

C     Terminate access to the Vdata interface.
      status = vfend(file_id)

C     Close the HDF file.
      status = hclose(file_id)

      end
```

4.4 Creating and Writing to Single-Field Vdatas: VHstoredata and VHstoredatam

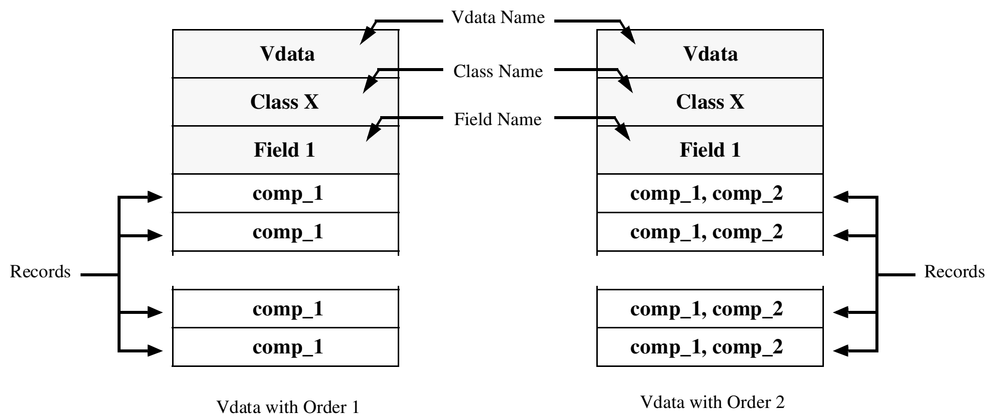
There are two methods of writing vdatas that contain one field per record. One requires the use of VS routines and the other involves the use of **VHstoredata** or **VHstoredatam**, two high-level routines that encapsulate several VS routines into one.

The high-level VH routines are useful for writing vdatas when it is only necessary to write one field per vdata and complete information about each vdata to be written is available. If you cannot provide full information about the vdata the VS routines described in the next section must be called.

Figure 4c on page 108 shows two examples of single-field vdatas. The fields can be single-component or multi-component fields, that is, they may contain one or more components of the same data type.

FIGURE 4c

Single- and Multi-Component Vdatas



VHstoredata writes a vdata with one single-component field. **VHstoredatam** writes a vdata with one multi-component field. In both cases the following steps are involved:

1. Open the file.
2. Initialize the VS interface.
3. Create the vdata.
4. Terminate access to the VS interface.
5. Close the file.

These steps correspond to the following sequence of function calls:

C:

```
file_id = Hopen(filename, file_access_mode, n_dds);
status = Vstart(file_id);
vdata_ref = VHstoredata (file_id, fieldname, buf, n_records,
    data_type, vdata_name, vdata_class);

    OR

vdata_ref = VHstoredatam (file_id, fieldname, buf, n_records,
    data_type, vdata_name, vdata_class, order);
status = Vend(file_id);
status = Hclose(file_id);
```

FORTRAN:

```
file_id = hopen(filename, file_access_mode, n_dds)
status = vfstart(file_id)
vdata_ref = vhfsd (file_id, fieldname, buf, n_records,
    data_type, vdata_name, vdata_class)
status = vfend(file_id)
status = hclose(file_id)

    OR

vdata_ref = vhfsdm (file_id, fieldname, buf, n_records,
    data_type, vdata_name, vdata_class, order)
status = vfend(file_id)
status = hclose(file_id)
```

The first seven parameters of **VHstoredata** and **VHstoredatam** are the same. The `file_id` parameter is the file identifier returned by **Hopen**. The `fieldname` parameter is the name of the single field. The `buf`, `n_records` and `data_type` parameters contain the contents, number of records and data type of the vdata and the `vdata_name` and `vdata_class` parameters specify the name and class of the vdata. The `order` parameter of **VHstoredatam** specifies the order of the multi-component field. The maximum length of the vdata name is given by the `VSNAMELENMAX` definition in the "hlimits.h" header file. Both routines return the reference number of the newly-created vdata or `FAIL` (or `-1`) if the create operation was unsuccessful.

The parameters of **VHstoredata** and **VHstoredatam** are further described in the following table.

TABLE 4C

VHstoredataand VHstoredatam Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
VHstoredata (vhfsd/vhfscd)	file_id	int32	integer	File identifier.
	fieldname	char *	character* (*)	String containing the name of the field.
	buf	uint8	integer (*)	Buffer containing the data to be stored.
	n_records	int32	integer	Number of records to create in the vdata.
	data_type	int32	integer	Number type of the stored data.
	vdata_name	char *	character* (*)	Name of the vdata.
	vdata_class	char *	character* (*)	Vdata class.
VHstoredatam (vhfsdm/ vhfscdm)	file_id	int32	integer	File identifier.
	fieldname	char *	character* (*)	String containing the name of the single field.
	buf	uint8 *	integer (*)	Buffer containing the data to be stored.
	n_records	int32	integer	Number of records to create in the vdata.
	data_type	int32	integer	Number type of the stored data.
	vdata_name	char *	character* (*)	Name of the vdata.
	vdata_class	char *	character* (*)	Class describing the type of vdata.
	order	int32	character* (*)	Number of components to store in the field.

EXAMPLE 2.

Creating Single-Field Vsets

In these examples two vdatas are created. The first vdata has five records with one field of order 1 and is created from a 5 x 1 array in memory. The second vdata has six records with one fields of order 4 and is created from a 6 x 4 array in memory.

C:

```
#include "hdf.h"

main( )
{

int32 file_id, vdata_id, vdata_ref1, vdata_ref2, status;
uint8 vset1_data[5] = {1, 2, 3, 4, 5};
uint8 vset2_data[6][4] = {{1, 2, 3, 4}, {2, 4, 6, 8},
                          {3, 6, 9, 12}, {4, 8, 12, 16},
                          {5, 10, 15, 20}, {6, 12, 18, 24}};

/* Create the HDF file. */
```

```
file_id = Hopen("Example2.hdf", DFACC_CREATE, 0);

/* Initialize the VS interface. */
status = Vstart(file_id);

/* Create the first vdata and populate it with data
* from the vset1_data array. */

vdata_ref1 = VHstoredata(file_id, "Vset1 data", (uint8 *)vset1_data, \
                5, DFNT_UINT8, "Second Vset", "5x1 array");

/*
* Create the second vdata and populate it with data
* from the vset2_data array.
*/

vdata_ref2 = VHstoredatam(file_id, "Vset2 data", (uint8 *)vset2_data, \
                6, DFNT_UINT8, "Third Vset", "6x4 array", 4);

/* Terminate access to the VS interface. */
status = Vend(file_id);

/* Close the HDF file. */
status = Hclose(file_id);

}
```

FORTRAN:

```
     PROGRAM VSET CREATE

      integer*4 file_id, vdata_ref1, vdata_ref2
      integer vset_data1(5), vset_data2(6, 4), i
      integer hopen, vhfsd, vhfsdm, hclose

C     DFACC_CREATE is defined in 'hdf.inc'. DFNT_INT32 is
C     defined in 'hntdefs.h'.
      integer DFACC_CREATE, DFNT_INT32
      parameter (DFACC_CREATE = 4, DFNT_INT32 = 24)

C     Generate the vset_data1 data.
      do 10 i = 1, 5
        vset_data1(i) = i
10    continue

C     Generate the vset_data2 data.
      do 20 i = 1, 6
        do 30 j = 1, 4
          vset_data2(i, j) = i + j
30      continue
20    continue

C     Create the HDF file.
      file_id = hopen('Example2.hdf', DFACC_CREATE, 0)

C     Initialize the VS interface.
      status = vfstart(file_id)

C     Create the first vdata and populate it with data
C     from the vset1_data array.
      vdata_ref1 = vhfsd(file_id, 'Vset1 data', vset_data1,
     +               5, DFNT_INT32, 'First Vset', '5x1 array')
```

```
C     Create the second vdata and populate it with data
C     from the vset2_data array.
      vdata_ref2 = vhfsdm(file_id, 'Vset2 data', vset_data2,
     +                6, DFNT_INT32, 'Second Vset', '6x4 array', 4)

C     Terminate access to the VS interface.
      status = vfend(file_id)

C     Close the HDF file.
      status = hclose(file_id)

      end
```

4.5 Creating and Writing to Multi-Field Vdatas

There are three steps involved in creating vdatas with more than one field, or ***general vdatas***: define the vdata, write the data to the vdata then write the vdata to the file. These steps are usually executed within a single program, although it is also possible to define an empty vdata in anticipation of writing data to it at a later time.

4.5.1 Creating Vdatas

Creating an empty vdata involves the following steps:

1. Open a file.
2. Initialize the VS interface.
3. Create the new vdata.
4. Assign a vdata name. (optional)
5. Assign a class. (optional)
6. Define the fields.
7. Set the interlace mode.
8. Dispose of the vdata identifier.
9. Terminate access to the VS interface.
10. Close the file.

Like the high-level VH interface, the VS interface does not retain default settings from one operation to the next or from one file to the next. Each time a vdata is created, its definitions must be explicitly reset.

To create a multi-component vdata, the calling program must contain the following:

C:
```
file_id = Hopen(filename, file_access_mode, n_dds);
status = Vstart(file_id);
vdata_id = VSattach(file_id, vdata_ref, vdata_access_mode);
status = VSsetname(vdata_id, vdata_name);
status = VSsetclass(vdata_id, vdata_class);
status = VSfdefine(vdata_id, fieldname, data_type, order);
status = VSsetfields(vdata_id, fieldname_list);
status = VSsetinterlace(vdata_id, interlace_type);
status = VSdetach(vdata_id);
status = Vend(file_id);
status = Hclose(file_id);
```

FORTRAN:
```
file_id = hopen(filename, file_access_mode, n_dds)
status = vfstart(file_id)
```

```
vdata_id = vsfatch(file_id, vdata_ref, vdata_access_mode)
status = vsfsnam(vdata_id, vdata_name)
status = vsfscls(vdata_id, vdata_class)
status = vsffdef(vdata_id, fieldname, data_type, order)
status = vsfsfld(vdata_id, fieldname_list)
status = vsfsint(vdata_id, interlace_type)
status = vsfdtch(vdata_id)
status = vfend(file_id)
status = hclose(file_id)
```

In the routines that follow, `vdata_id` is the vdata identifier returned by **VSattach.**

4.5.1.1 Creating New Vdatas with VSattach

When a new vdata is created, the value of the **VSattach** parameter `vdata_ref` must be `-1` and the value of the `access_mode` parameter must be `"w"`.

4.5.1.2 Assigning a Vdata Name and Class: VSsetname and VSsetclass

VSsetname assigns a name to a vdata object. If not explicitly named by a call to **VSsetname**, the name of the vdata is set by default to `NULL`. Names may be assigned and reassigned at any time after the vdata is created. The parameter `vdata_name` contains the name to be assigned to the vdata.

VSsetclass assigns a class to a vdata. If **VSsetclass** is not called, the vdata's class is set by default to `NULL`. As with vdata names, the class may be assigned and reassigned any time after the vdata is created. The parameter `vdata_class` contains the class name to be assigned to the vdata.

The parameters for **VSsetname** and **VSsetclass** are further defined below. (See Table 4E on page 113.)

4.5.1.3 Defining a Field Within Vdatas: VSfdefine

VSfdefine defines a field within a newly-created vdata. Each **VSfdefine** call assigns the name contained in the `fieldname` argument, the data type contained in the `data_type` argument and the order contained in the `order` argument to one new field. Fields will appear in the vdata in the order they are defined. Once data is written to a vdata, field definitions may not be modified or deleted.

The VS interface also provides certain ***predefined fields***. (See Table 4D.) A predefined field is assumed to have a specific name, data type and order, so there is no need to call **VSfdefine** to define a predefined field. Some applications may require the use of pre-defined fields in vdatas. (For instance, NCSA Polyview requires the coordinates of vertices to have the predefined field names "PX", "PY" and "PZ".)

The parameters of **VSfdefine** are further described below. (See Table 4E on page 113.)

TABLE 4D **Predefined Data Types and Field Names for Vdata Fields**

Data Type	Coordinate Point Field Names			Normal Component Field Names		
	x-coordinate	y-coordinate	z-coordinate	x-component	y-component	z-component
float	PX	PY	PZ	NX	NY	NZ
integer	IX	IY	IZ	None	None	None

4.5.1.4 Initializing the Fields for Write Access: VSsetfields

VSsetfields initializes write access to the fields in a vdata. It is used to specify the fields to be written and the order they are to be written in . The parameter `fieldname_list` is a comma-separated list of all the fields that will be accessed during the write operation. It may contain any fieldname previously defined by **VSfdefine**, but it may also contain names of predefined fields.

VSsetfields must be called prior to any write operation and may only be called once for each vdata. Any field not named in the `fieldname_list` parameter is removed from the vdata. The combined width of the fields must be less than 32,767 bytes. If an attempt is made to create a larger record than this, **VSsetfields** will fail and an error condition will result.

The parameters of **VSsetfields** are further described below. (See Table 4E on page 113.)

4.5.1.5 Specifying the Interlace Mode: VSsetinterlace

The VS interface supports two modes of interlacing: ***file interlacing*** and ***buffer interlacing***. File interlacing determines how data is stored in a vdata and buffer interlacing determines how data is stored in memory. The VS interface can write data from a buffer to a file in an interlaced or non-interlaced manner. It can also read data from a file in an interlaced or non-interlaced manner.

The **VSread** and **VSwrite** routines set the file interlace mode. The **VSwrite** routine will be discussed in Section 4.5.2.2 on page 115 and the **VSread** routine will be discussed in Section 4.6.3 on page 123.

VSsetinterlace sets the file interlacing mode for a vdata. Setting the parameter `interlace` to `FULL_INTERLACE` fills the vdata by record, whereas specifying `NO_INTERLACE` fills the vdata by field. (See Figure 4d.) For multi-component fields, all components are treated as a single field. As with file interlacing, the default vdata interlace mode is `FULL_INTERLACE` because it is more efficient to write complete records than it is to write fields if the file and memory interlace modes are the same, although both require the same amount of disk space.

FIGURE 4d **Interlaced and Non-Interlaced Vdata Contents**

Vdata			
Mixed_Data_Type			
Temp	**Height**	**Speed**	**Ident**
1.11	**1**	**11.11**	**A**
2.22	**2**	**22.22**	**B**
3.33	**3**	**33.33**	**C**

File Interlacing: FULL_INTERLACE

Vdata			
Mixed_Data_Type			
Temp	**1.11**	**2.22**	**3.33**
Height	**1**	**2**	**3**
Speed	**11.11**	**22.22**	**33.33**
Ident	**A**	**B**	**C**

File Interlacing: NO_INTERLACE

VSsetinterlace can only be used for operations on new vdatas as the interlacing cannot be changed once the data has been written to a vdata object. Also, although records in a fully interlaced vdata can be written with one or more write operations, all records in a non-interlaced vdata must be written at the same time.

The parameters of **VSsetinterlace** are further described in the following table.

TABLE 4E **VSsetname, VSsetclass, VSfdefine, VSsetfields and VSsetinterlace Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	

VSsetname (vsfsnam)	vdata_id	int32	integer	Vdata identifier.
	vdata_name	char *	character* (*)	Vdata name.
VSsetclass (vsfscls)	vdata_id	int32	integer	Vdata identifier.
	vdata_class	char *	character* (*)	Vdata name.
VSfdefine (vsffdef)	vdata_id	int32	integer	Vdata identifier.
	fieldname	char *	character* (*)	Name of the field to be defined.
	data_type	int32	integer	Type of the field data.
	order	int32	integer	Order of the new field.
VSsetfields (vsfsfld)	vdata_id	int32	integer	Vdata identifier.
	fields	char *	character* (*)	Names of the vdata fields to be accessed.
VSsetinterlace (vsfsint)	vdata_id	int32	integer	Vdata identifier.
	interlace	int32	integer	Interlace mode.

4.5.2 Writing Data to Vdatas

Writing to a vdata requires the following steps:

1. Open a file.
2. Initialize the VS interface.
3. Access the vdata.
4. Seek to the target record.
5. Write the data.
6. Dispose of the vdata identifier.
7. Terminate access to the VS interface.
8. Close the file.

To write data to a vdata, the calling program must contain the following sequence of calls:

C:

```
file_id = Hopen(filename, file_access_mode, n_dds);
status = Vstart(file_id);
vdata_id = VSattach(file_id, vdata_ref, vdata_access_mode);
status = VSsetfields(vdata_id, *fields);
record_pos = VSseek(vdata_id, record_index);
num_of_recs = VSwrite(vdata_id, *databuf, n_records, interlace);
status = VSdetach(vdata_id);
status = Vend(file_id);
status = Hclose(file_id);
```

FORTRAN:

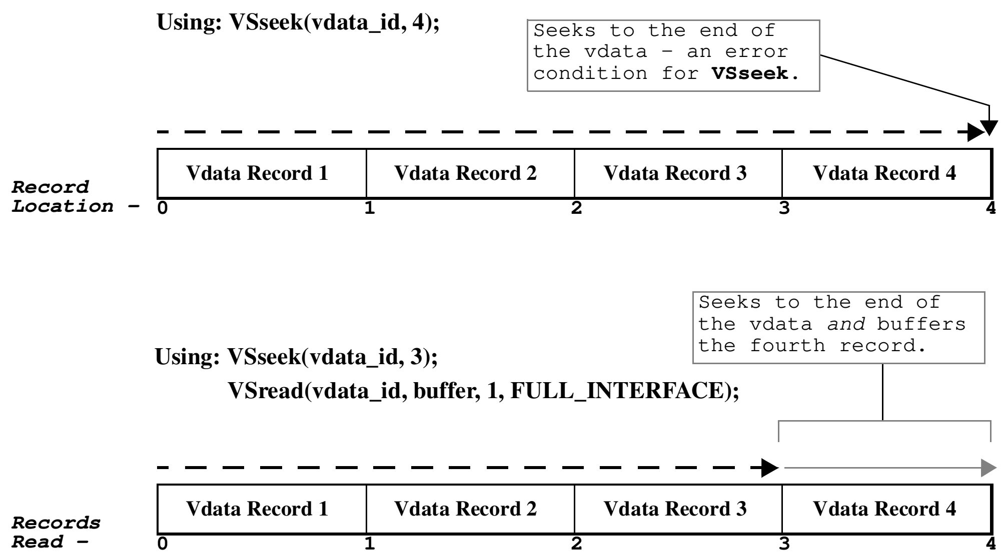
```
file_id = hopen(filename, file_access_mode, n_dds)
status = vfstart(file_id)
vdata_id = vsfatch(file_id, vdata_ref, vdata_access_mode)
status = vsfsfld(vdata_id, *fields);
record_pos = vsfseek(vdata_id, record_index);
num_of_recs = vsfwrit(vdata_id, *databuf, n_records, interlace)
status = vsfdtch(vdata_id)
status = vfend(file_id)
status = hclose(file_id)
```

4.5.2.1 Resetting the Current Position Within Vdatas: VSseek

VSseek provides a mechanism for random-access write operations within fully-interlaced vdatas. Random-access writes are not available for non-interlaced vdatas. The parameter `record_index` is the position of the record to be written. The position of the first record in a vdata is specified by `record_index=0`: any vdata operation will be performed on this vdata before **VSseek** is called.

One thing that should be kept in mind while using **VSseek** is that it has been designed for the purpose of *overwriting* data, not *appending* data. Figure 4e illustrates a situation where this aspect of **VSseek** can be misused with unexpected results and how **VSread** is called to correctly place the record pointer at the end of the vdata object for appending.

FIGURE 4e **Setting the Record Pointer at the End of a Vdata Object**



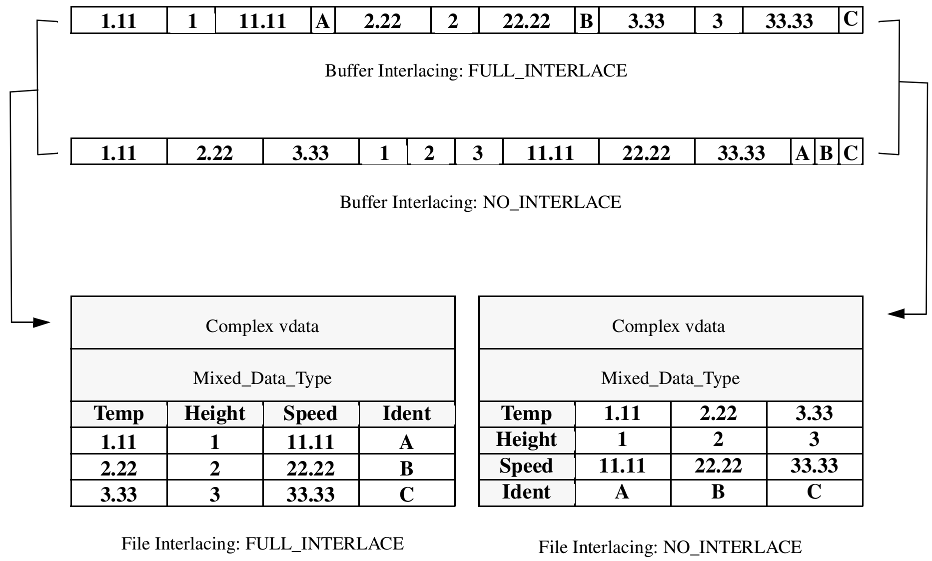
In this figure we have a vdata that consists of only four records. Using **VSseek** to seek to the end of the fourth record by setting the `record` parameter to `4` results in an error condition. To avoid this situation, we set the `record` parameter to `3` in order to set the current record pointer to the end of the third record. We then use **VSread** to read the contents of the fourth record in a buffer and move the current record pointer to the end of the fourth record. The contents of the buffer can then be discarded. This method can be generalized to a vdata of any number of records; seek to the last record with **VSseek**, then read the last record using **VSread**.

4.5.2.2 Writing to a Vdata: VSwrite

VSwrite writes buffered data to the vdata in the specified HDF file. The parameter `databuf` is a buffer containing records to store in the vdata. The `n_records` parameter is the number of records to be written.

The array `databuf` is assumed to be organized in memory as specified by `interlace`. Selecting `FULL_INTERLACE` indicates that the array in memory is organized by record, whereas `NO_INTERLACE` indicates that the array is organized by field. (See Figure 4f.) Notice that *file interlacing* is set by **VSsetinterlace** when the vdata is created, whereas the *buffer interlacing* is set by **VSwrite** when data is written to file. **VSwrite** will write interlaced or non-interlaced data to a file as an interlaced or non-interlaced vdata. However, multiple write operations can only be used on fully-interlaced vdatas.

FIGURE 4f **Writing Interlaced or Non-Interlaced Buffers into Interlaced or Non-interlaced Vdatas**



The data in `databuf` is assumed to contain the exact amount of data in the order needed to fill the fields defined in the last call to **VSsetfields**. Because **VSwrite** writes the contents of `databuf` contiguously to the vdata, any "padding" due to record alignment must be removed before attempting to write from `databuf` to the vdata. For more information on removing alignment padding see Section 4.5.2.3 on page 118.

It should be remembered that **VSwrite** writes whole records, not individual fields. If a modification to one field within a previously-written record is needed, the contents of the record must first be preserved by reading it using **VSread**, which will be described in Section 4.6.3 on page 123, updating the field then writing it using **VSwrite**.

To create an empty vdata either **VSwrite**, **VSsetname** and **VSsetclass** must be called before **VSdetach**. If **VSwrite** is not called the vdata created will be an empty vdata. If **VSsetfields** is called and **VSwrite**, **VSsetname** or **VSsetclass** is not called, the vdata will not be written.

TABLE 4F **VSseek and VSwrite Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
VSseek (**vsfseek**)	vdata_id	int32	integer	Vdata identifier.
	record	int32	integer	Record number to seek to.
VSwrite (**vsfwrit**)	vdata_id	int32	integer	Vdata identifier.
	databuf	char *	character* (*)	Buffer for records to be stored.
	n_records	int32	integer	Number of records to be stored.
	interlace	int32	integer	Interlace mode of the buffered data.

EXAMPLE 3. **Writing Data to a New Vdata**

In these examples, the VS interface is used to create a vdata containing ten rows, each comprised of one 16-bit integer field of order 3. The vdata is named "Example Vdata", the vdata class is "Example Vdata" and the field is named "Field Entries".

C:

```
#include “hdf.h”

#define FIELD_NAME “Field Entries”
#define NUMBER_OF_ROWS 10
#define ORDER 3

main( )
{

int32 file_id, vdata_id, status, num_of_elements;
int16 vdata_buf[NUMBER_OF_ROWS * ORDER], i;

/* Open the HDF file. */
file_id = Hopen(“Example3.hdf”, DFACC_CREATE, 0);

/* Initialize HDF for subsequent the vgroup/vdata access. */
status = Vstart(file_id);

/* Create a new vdata. */
vdata_id = VSattach(file_id, -1, “w”);

/* Define the field data name, type and order. */
status = VSfdefine(vdata_id, FIELD_NAME, DFNT_INT16, ORDER);

/* Specify the field(s) that will be written to. */
status = VSsetfields(vdata_id, FIELD_NAME);

/* Generate the Vset data. */
for (i = 0; i < NUMBER_OF_ROWS * ORDER; i+=ORDER) {
   vdata_buf[i] = i;
   vdata_buf[i + 1] = i + 1;
   vdata_buf[i + 2] = i + 2;
}

/* Write the data to the Vset. */
num_of_elements = VSwrite(vdata_id, (char *)vdata_buf, NUMBER_OF_ROWS,
                          FULL_INTERLACE);

/* Set the name and class. */
status = VSsetname(vdata_id, “Example Vdata”);
status = VSsetclass(vdata_id, “Example Vdata”);

/* Terminate access to the vdata. */
status = VSdetach(vdata_id);

/* Terminate access to the Vset interface and close the file. */
status = Vend(file_id);
status = Hclose(file_id);

}
```

FORTRAN:

```
     PROGRAM WRITE VDATA

      integer file_id, vdata_id
      integer status
```

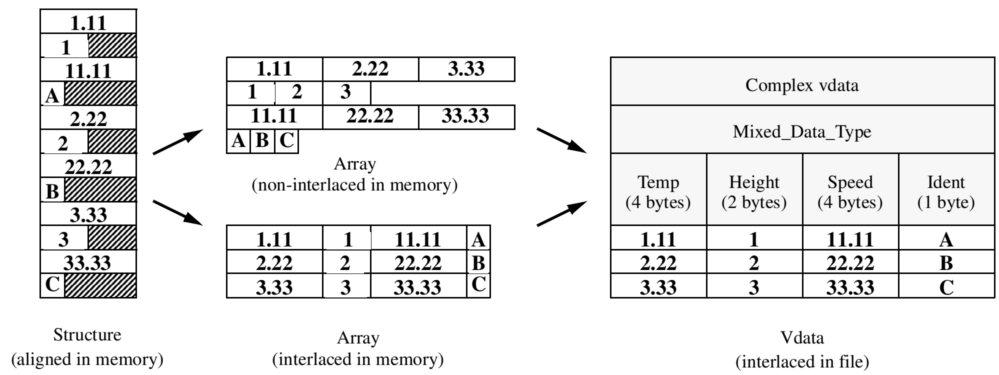
```
      integer vdata_buf(30), i, number_of_rows, num_of_elements
      integer hopen, vsfatch, vsffdef, vsfsfld
      integer vsfwrit, vsfsnam, vsfscls, vsfdtch
      integer hclose, vfstart, vfend

C     DFACC_CREATE, DFNT_INT16 and FULL_INTERLACE are defined
C     in hdf.inc.
      integer DFACC_CREATE, DFNT_INT16, FULL_INTERLACE
      parameter (DFACC_CREATE = 4,
     +           DFNT_INT16 = 22,
     +           FULL_INTERLACE = 0)

      character field_name *13
      parameter (field_name = 'Field Entries')
      parameter (number_of_rows = 10)

C     Create an HDF file with full access.
      file_id = hopen('Example3.hdf', DFACC_CREATE, 0)

C     Initialize HDF for subsequent vgroup/vdata access.
      status = vfstart(file_id)

C     Create a new vdata.
      vdata_id = vsfatch(file_id, -1, 'w')

C     Define the field data name, type and order.
      status = vsffdef(vdata_id, field_name, DFNT_INT16, 3)

C     Specify the field that will be written to.
      status = vsfsfld(vdata_id, field_name)

C     Generate the Vset data.
      do 10 i = 1, number_of_rows * 3, 3
        vdata_buf(i) = i+1
        vdata_buf(i + 1) = i+2
        vdata_buf(i + 2) = i+3
10    continue

C     Set the name and class.
      status = vsfsnam(vdata_id, 'Example Vdata')
      status = vsfscls(vdata_id, 'Example Vdata')

C     Write the data to the Vset.
      num_of_elements = vsfwrit(vdata_id, vdata_buf, number_of_rows *
3,
     +                FULL_INTERLACE)

C     Terminate access to the vdata.
      status = vsfdtch(vdata_id)

C     Terminate access to the Vset interface and close the file.
      status = vfend(file_id)
      status = hclose(file_id)

      end
```

4.5.2.3 Filling Records of Mixed Field Data Types

Storing fields of mixed data types is an efficient use of disk space and is useful in applications that use structures. While data structures in memory containing fields of variable lengths often contain padding or alignment bytes, data stored in a vdata does not contain padding. This is true for both fully-interlaced and non-interlaced data. Because of this, when variable-length field types are used it is not generally possible to write data directly from a structure in memory into a vdata in a file with a **VSwrite** call. Instead, the data from the structure must first be written to a temporary buffer array, removing all alignment bytes. This packed array is then written to a vdata using **VSwrite**. (See Figure 4b.)

FIGURE 4g

Removing Alignment Bytes When Writing Data From a C Structure to a Vdata



EXAMPLE 4.

Writing Packed Vdata Data to an HDF File

The following examples write packed data into a vdata. Note that while the **memcpy** routine is called for each field, the **bcopy** routine may be used instead.

Fortran-77 does not support record structures so there is no standard way to do packing as in C. The Fortran-77 example will work in many platforms but it is not guaranteed to be 100% portable.

C:

```
#include <stdio.h>
#include “hdf.h”

#define NRECORDS 20

#define FIELD_1 “Temp”
#define FIELD_2 “Height”
#define FIELD_3 “Speed”
#define FIELD_4 “Ident”
#define FIELD_NAMES  “Temp,Height,Speed,Ident”

main( )
{

struct {
   float32  temp;
   int16    height;
   float32  speed;
   char     ident;
} source[NRECORDS];

int32file_id, vdata_id;
uint8 *pntr;
uchar8 *databuf;
int i;
int32 status;
```

```
int32 msize = 0;

/* Create the HDF file. */
file_id = Hopen("Example4.hdf", DFACC_CREATE, 0);

/* Initialize the Vset interface. */
status = Vstart(file_id);

/* Create a new vdata. */
vdata_id = VSattach(file_id, -1, "w");

/* Define the field to write. */
status = VSfdefine(vdata_id, FIELD_1, DFNT_FLOAT32, 1);
status = VSfdefine(vdata_id, FIELD_2, DFNT_INT16, 1);
status = VSfdefine(vdata_id, FIELD_3, DFNT_FLOAT32, 1);
status = VSfdefine(vdata_id, FIELD_4, DFNT_CHAR8, 1);

/* Set the vdata name. */
status = VSsetname(vdata_id, "Example Vset Name");

/* Set the vdata class. */
status = VSsetclass(vdata_id, "Example Vset Class");

/* Set the field names. */
status = VSsetfields(vdata_id, FIELD_NAMES);

databuf = (uint8 *) malloc(((2 * sizeof(float32))
           + sizeof(int16) + sizeof(char)) * NRECORDS);

pntr = databuf;

for (i = 0; i < NRECORDS; i++) {
   source[i].temp = 1.11 * (i+1);
   memcpy(pntr, &source[i].temp, sizeof(float32));
   pntr += sizeof(float32);

   source[i].height = i+1;
   memcpy(pntr, &source[i].height, sizeof(int16));
   pntr += sizeof(int16);

   source[i].speed = 1.11 * (i+1);
   memcpy(pntr, &source[i].speed, sizeof(float32));
   pntr += sizeof(float32);

   source[i].ident = 'A'+i;
   memcpy(pntr, &source[i].ident, sizeof(char));
   pntr += sizeof(char);
}

/* Write the data to the Vset object. */
status = VSwrite(vdata_id, databuf, NRECORDS, FULL_INTERLACE);

/*
* Terminate access to the vdata, the VS interface
* and the HDF file.
*/
status = VSdetach(vdata_id);
status = Vend(file_id);
status = Hclose(file_id);

}
```

FORTRAN:

```
      PROGRAM DATA PACK

       integer hopen, vsfatch, vsffdef, vsfsnam
       integer vsfscls, vsfsfld, vsfwrit, vsfdtch
       integer hclose, vfstart

C      Parameter definitions
       integer*4 DFACC_CREATE, DFNT_FLOAT32, DFNT_INT16, DFNT_CHAR8
       integer*4 FULL_INTERLACE
       parameter (DFACC_CREATE = 4,
      +           DFNT_FLOAT32 = 5,
      +           DFNT_INT16 = 22,
      +           DFNT_CHAR8 = 4,
      +           FULL_INTERLACE = 0)

C      Example parameters.
       integer NRECORDS
       parameter (NRECORDS = 20)

C      Vdata fields.
       real temp
       integer*2 height
       real speed

C      Vdata field equivalence names for packing.
       character ctemp*4
       character cheight*2
       character cspeed*4
       equivalence (temp, ctemp), (height, cheight), (speed, cspeed)

C      Packing buffer with size (4+2+4+1)*NRECORDS = 220.
       character buffer*220

C      More local variables.
       integer i, bufptr, status
       integer*4 file_id, vdata_id
       character*20 idents /"ABCDEFGHIJKLMNOPQRST"/

C      Create the file.
       file_id = hopen('Example4.hdf', DFACC_CREATE, 0)

C      Initialize the interface.
       status = vfstart(file_id)

C      Create a new vdata.
       vdata_id = vsfatch(file_id, -1, 'w')

C      Define the fields to write.
       status = vsffdef(vdata_id, 'Temp', DFNT_FLOAT32, 1)
       status = vsffdef(vdata_id, 'Height', DFNT_INT16, 1)
       status = vsffdef(vdata_id, 'Speed', DFNT_FLOAT32, 1)
       status = vsffdef(vdata_id, 'Ident', DFNT_CHAR8, 1)

C      Set the vdata name.
       status = vsfsnam(vdata_id, 'Example Vset Name')

C      Set the vdata class.
       status = vsfscls(vdata_id, 'Example Vset Class')

C      Set the field names.
       status = vsfsfld(vdata_id, 'Temp,Height,Speed,Ident')
```

```
C     Pack NRECORDS of data into buffer.
      bufptr = 1

      do 10 i = 1, NRECORDS
C       Pack temp data.
        temp = 1.1 * i
        buffer(bufptr:bufptr+3) = ctemp
        bufptr = bufptr + 4

C       Pack height data.
        height = i
        buffer(bufptr:bufptr+1) = cheight
        bufptr = bufptr + 2

C       Pack speed data.
        speed = 11.1 * i
        buffer(bufptr:bufptr+3) = cspeed
        bufptr = bufptr + 4

C       Pack ident data.
        buffer(bufptr:bufptr) = idents(i:i)
        bufptr = bufptr + 1
10    continue

      status = vsfwrit(vdata_id, buffer, NRECORDS, FULL_INTERLACE)

C     Terminate access to the vdata object, the interface
C     and the file.
      status = vsfdtch(vdata_id)
      status = vfend(file_id)
      status = hclose(file_id)

      end
```

4.6 Reading from Vdatas

Reading from vdatas is more complicated than writing to vdatas, as it usually involves searching for a particular vdata, as well as searching within the vdata, before actually reading data. The process of reading from vdatas can be summarized in three steps:

1. Identify the appropriate vdata in the file.
2. Obtain information about the vdata.
3. Read in the desired data**.**

The last step will be covered first as the vdata of interest is usually known, therefore the first two steps are not needed.

These three basic steps can be expanded into the following:

1. Open a file.
2. Initialize the VS interface.
3. Access the vdata.
4. Optionally seek to the records to access.
5. Initialize the fields to access.
6. Read the data.
7. Detach from the vdata.

8. Terminate access to the VS interface.
9. Close the file.

The calling program must contain the following sequence of calls:

C:
```
file_id = Hopen(filename, file_access_mode, n_dds);
status = Vstart(file_id);
vdata_id = VSattach(file_id, vdata_ref, vdata_access_mode);
record_pos = VSseek(vdata_id, record_index);
status = VSsetfields(vdata_id, fieldname_list);
status = VSread(vdata_id, *databuf, n_records, interlace);
status = VSdetach(vdata_id);
status = Vend(file_id);
status = Hclose(file_id);
```

FORTRAN:
```
file_id = hopen(filename, file_access_mode, n_dds)
status = vfstart(file_id)
vdata_id = vsfatch(file_id, vdata_ref, vdata_access_mode)
record_pos = vsfseek(vdata_id, record_index)
status = vsfsfld(vdata_id, fieldname_list)
status = vsfread(vdata_id, *databuf, n_records, interlace)
status = vsfdtch(vdata_id)
status = vfend(file_id)
status = hclose(file_id)
```

4.6.1 Selecting the Record to Begin Reading from: VSseek

VSseek is described in Section 4.5.2.1 on page 114.

4.6.2 Selecting the Fields to be Read: VSsetfields

VSsetfields establishes access to the fields targeted for the next read operation. The argument `fieldname_list` is a comma-separated string of the target field names with no white space

The order the field names occur in `fieldname_list` is the order in which the fields will be read. For example, assume that a vdata contains fields named “A, B, C, D, E, F” and they are stored in alphabetical order. The following declarations demonstrate how to use `fieldname_list` to read a single field, a collection of random fields and a series of fields in reverse order;

- Single field: `fieldname_list = "B".`
- Collection of fields: `fieldname_list = "A,E".`
- Reverse order: `fieldname_list = "F,E,D,C,B,A".`

4.6.3 Reading from the Current Vdata: VSread

VSread sequentially retrieves data from the records in a vdata. The parameter `databuf` is the buffer to store the retrieved data, `n_records` is the number of records to retrieve and `interlace` specifies the interlace mode (`FULL_INTERLACE` or `NO_INTERLACE`) to be used in the contents of `databuf`. **VSread** returns the total number of records read if successful and `FAIL` (or `-1`) otherwise.

Prior to the first **VSread** call, **VSsetfields** must be called.

If a **VSread** call is successful, the data returned in `databuf` is formatted according to the interlace mode specified by `interlace` and the data fields appear in the order specified in the last call to **VSsetfields** for that vdata. To retrieve an arbitrary record from a vdata, use **VSseek** to specify the record position before calling **VSread**.

TABLE 4G

VSseek, VSsetfields and VSread Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
VSseek (**vsfseek**)	vdata_id	int32	integer	Vdata identifier.
	record	int32	integer	Record number to seek to.
VSsetfields (**vsfsfld**)	vdata_id	int32	integer	Vdata identifier.
	fields	char *	character* (*)	Names of vdata fields to be accessed.
VSread (**vsfread**)	vdata_id	int32	integer	Vdata identifier.
	databuf	char *	character* (*)	Buffer for the retrieved data.
	n_records	int32	integer	Number of records to be retrieved.
	interlace	int32	integer	Interlace mode of the buffered data.

VSsetfields and **VSread** may be called several times to read from the same vdata. However, note that **VSread** operations are sequential. Thus, in the following code segment, the first call to **VSread** returns ten "A" data values from the first ten elements in the vdata, while the second call to **VSread** returns ten "B" data values from the second ten elements (elements 10 to 19) in the vdata.

```
VSsetfields(vs, "A");
VSread(vs, bufA, 10, interlace);

VSsetfields(vs, "B");
VSread(vs, bufB, 10, interlace);
```

To read the first ten "B" data values, the access routine **VSseek** must be called to explicitly position the read pointer back to the position of the first record. The following code segment reads the first ten "A" and "B" values into two separate float arrays `bufA` and `bufB`.

```
VSsetfields(vs, "A");
VSread(vs, bufA, 10, interlace);

VSseek(vs, 0); /* seeks to first element */
VSsetfields(vs, "B");
VSread(vs, bufB, 10, interlace);
```

EXAMPLE 5.

Reading Data from a Vdata

In the following examples, the VS interface is used to read the vdata created in the Example 4. **VSgetid** is used to demonstrate a method of obtaining the reference id number of the first vdata in the HDF file.

C:

```
#include "hdf.h"
#define NRECORDS 20

main( )
{

char vdata_name[MAX_NC_NAME], vdata_class[MAX_NC_NAME], fields[60];
int32 file_id, vdata_id, status, num_of_elements;
int32 n_records, interlace, vdata_size, vdata_ref;
uint8 databuf[((2 * sizeof(float32)) + sizeof(char) \
```

```
            + sizeof(int16)) * NRECORDS];

/* Open the HDF file. */
file_id = Hopen("Example4.hdf", DFACC_RDONLY, 0);

/* Initialize the Vset interface. */
status = Vstart(file_id);

/*
* Get the reference number for the first vdata in
* the file.
*/
vdata_ref = -1;
vdata_ref = VSgetid(file_id, vdata_ref);

/* Attach to the first vdata in read mode. */
vdata_id = VSattach(file_id, vdata_ref, "r");

/* Get the list of field names. */
status = VSinquire(vdata_id, &n_records, &interlace, fields,
                        &vdata_size, vdata_name);

/* Get the class. */
status = VSgetclass(vdata_id, vdata_class);

/* Determine the fields that will be read. */
status = VSsetfields(vdata_id, fields);

/* Print the vdata information. */
printf("Current vdata name: %s \nCurrent vdata class: %s\n", \
                     vdata_name, vdata_class);

/* Read the data. */
num_of_elements = VSread(vdata_id, databuf, n_records,
                FULL_INTERLACE);

/* Detach from the vdata, close the vset interface and the file. */
status = VSdetach(vdata_id);
status = Vend(file_id);
status = Hclose(file_id);

}
```

FORTRAN:

```
     PROGRAM DATA READ

      character vdata_name*30, vdata_class*30, fields*60
      character databuf*40
      integer file_id, vdata_ref, vdata_id
      integer*4 n_records, interlace, vdata_size
      integer status
      integer hopen, vsfgid, vsfatch, vsfinq, vsfgcls
      integer vsfsfld, vsfread, vsfdtch, hclose, vfstart, vfend

C     DFACC_RDONLY and FULL_INTERLACE are defined in hdf.inc.
      integer*4 DFACC_RDONLY, FULL_INTERLACE
      parameter (DFACC_RDONLY = 1,
     +           FULL_INTERLACE = 0)

C     Open an HDF file with read-only access.
      file_id = hopen('Example4.hdf', DFACC_RDONLY, 0)
```

```
C     Initialize the Vset interface.
      status = vfstart(file_id)

C     Get the reference number for the first Vdata in
C     the file.
      vdata_ref = -1
      vdata_ref = vsfgid(file_id, vdata_ref)

C     Attach to the first Vdata in read mode.
      vdata_id = vsfatch(file_id, vdata_ref, 'r')

C     Get the list of field names.
      status = vsfinq(vdata_id, n_records, interlace, fields,
     +               vdata_size, vdata_name)

C     Get the class.
      status = vsfgcls(vdata_id, vdata_class)

C     Determine the fields that will be read.
      status = vsfsfld(vdata_id, fields)

C     Print the Vdata information.
      print *, 'Current Vdata name: ', vdata_name
      print *, 'Current Vdata class: ', vdata_class

C     Read the data.
      status = vsfread(vdata_id, databuf, n_records, FULL_INTERLACE)

C     Detach from the Vdata, close the Vset interface and the file.
      status = vsfdtch(vdata_id)
      status = vfend(file_id)
      status = hclose(file_id)

      end
```

4.7 Searching for Vdatas in a File

There are several HDF routines that perform searches for a specific vdata in a file. In this section we introduce these four routines and in the following section methods for obtaining information about the members of a given vdata is described.

4.7.1 Finding All Vdatas that are Not Members of Vgroups: VSlone

A ***lone vdata*** is one that is not a member of a vgroup. The function **VSlone** returns the total number of lone vdatas. The parameter `ref_array` is an array allocated to hold the reference numbers of all lone vdatas in a file and the `maxsize` argument specifies the maximum size of `ref_array`. At most `maxsize` reference numbers will be returned in `ref_array`.

The space that should be allocated for `ref_array` is dependent upon on how many lone vdatas you expect to find. A size of 32,767 integers is adequate to handle any case. To use dynamic memory instead of allocating such a large array, first call **VSlone** with `maxsize` set to a small value like `0` or `1`, then use the returned value to allocate memory for `ref_array` to be passed to a subsequent call to **VSlone**.

The syntax of the **VSlone** function is as follows:

```
num_of_lone_vdatas = VSlone(file_id, ref_array, maxsize);
```

4.7.2 Searching for the Reference Number of a Vdata: VSgetid

VSgetid sequentially searches for vdata reference numbers in an HDF file. It returns the reference number for the vdata immediately following the vdata with the reference number `vdata_ref`. To initiate a search, **VSgetid** may be called using `vdata_ref = -1`. Doing so returns the reference number of the first vdata in the file. Searching past the last vdata in a file will return FAIL (or `-1`).

The syntax of the **VSgetid** function is as follows:

```
ref_num = VSgetid(file_id, vdata_ref);
```

4.7.3 Determining a Reference Number from a Vdata Name: VSfind

VSfind checks an HDF file for a vdata with the specified name and returns its reference number if it is found or `FAIL` (or `-1`) if not. The parameter `vdata_name` is the search key. Although there may be several identically named vdatas in a file, **VSfind** will only return the reference number of the first vdata in the file with the specified name.

The syntax of the **VSfind** function is as follows:

```
ref_num = VSfind(file_id, vdata_name);
```

4.7.4 Searching for a Vdata by Field Name: VSfexist

VSfexist queries a vdata for a set of specified field names and is often useful for locating vdata objects containing particular field names. Its return value is `1` if successful. The `fields` parameter is a string of comma-separated field names containing no white space - for example, "PX,PY,PZ".

The syntax of the **VSfexist** function is as follows:

```
status = VSfexist(vdata_id, fields);
```

To review, **VSfind** searches for a vdata with a specified name and returns its reference number if found. Given the reference number of the target vdata, **VSfexist** determines whether certain field names exist in it.

TABLE 4H **VSlone, VSgetid, VSfind and VSfexist Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
VSlone (vsflone)	file_id	int32	integer	File identifier.
	ref_array	int32 []	integer (*)	Buffer for a list of vdata reference numbers.
	maxsize	int32	integer	Maximum number of reference numbers to be buff- ered.
VSgetid (vsfgid)	file_id	int32	integer	File identifier.
	vdata_ref	int32	integer	Reference number of the vdata preceding the target vdata.
VSfind (vsffnd)	file_id	int32	integer	File identifier.
	vdata_name	char *	character* (*)	Name of the vdata to find.
VSfexist (vsfex)	vdata_id	int32	integer	Vdata identifier.
	fields	char *	character* (*)	Names of the fields to be queried.

EXAMPLE 6. **Locating a Vdata from a Specified Field Name**

The following examples search all vdatas in an HDF file for the first vdata containing the field names "Temp" and "Ident".

C:

```
#include “hdf.h”

#define FILE_NAME “Example4.hdf”

main( )
{

int32 file_id, vdata_ref, vdata_id, status;
int8 found_fields;

/* Open the HDF file. */
file_id = Hopen(FILE_NAME, DFACC_RDONLY, 0);

/* Initialize the vset interface. */
status = Vstart(file_id);

vdata_ref = -1;
found_fields = 0;

/* Attach to each vdata and search for the fields. */
while ((vdata_ref = VSgetid(file_id, vdata_ref)) != -1) {
   vdata_id = VSattach(file_id, vdata_ref, “r”);
   if ((status = VSfexist(vdata_id, “Temp,Speed,Height,Ident”))
            != FAIL) {
      found_fields = 1;
      break;
   } else status = VSdetach(vdata_id);
}

if (!found_fields)
    printf(“Fields TEMP and IDENT were not found.\n”);
else printf(“Fields TEMP and IDENT found in vdata id %d\n”,
        vdata_ref);

/* Detach from the vdata, close the Vset interface and the file. */
status = VSdetach(vdata_id);
status = Vend(file_id);
status = Hclose(file_id);

}
```

FORTRAN:

```
     PROGRAM SCAN VDATAS

      integer file_id, vdata_ref
      logical found_fields
      integer vdata_id, status
      integer hopen, vsfgid, vsfatch, vsfex, vsfdtch, hclose
      integer vfstart, vfend

C     DFACC_RDONLY is defined in hdf.inc.
      integer DFACC_RDONLY
      parameter (DFACC_RDONLY = 1)

C     Open an HDF file with full access.
      file_id = hopen(‘Example4.hdf’, DFACC_RDONLY, 0)
```

```
C     Initialize the Vset interface.
      status = vfstart(file_id)

      vdata_ref = -1
      found_fields = .FALSE.

C     Attach to each vdata and search for the fields.
30    vdata_ref = vsfgid(file_id, vdata_ref)
      if (vdata_ref .ne. -1) then
        vdata_id = vsfatch(file_id, vdata_ref, 'r')
        status = vsfex(vdata_id, 'Temp,Ident')
        if (status .eq. 1) then
          found_fields = .TRUE.
          go to 20
        else
          status = vsfdtch(vdata_id)
          go to 30
        end if
      end if

20    if (found_fields .eq. .FALSE.) then
        print *, 'Fields TEMP and IDENT were not found.'
      else
        print *, 'Fields TEMP and IDENT found in vdata id ',
     +           vdata_id
      end if

C     Detach from the vdata, close the Vset interface and the file.
      status = vsfdtch(vdata_id)
      status = vfend(file_id)
      status = hclose(file_id)

      end
```

4.8 Vdata Attributes

Version 4.1r1 of HDF includes the capability of assigning attributes to a vdata and/or a vdata field. The concept of attributes is fully explained in Chapter 3, titled *Scientific Data Sets (SD API)*. To review briefly: an attribute has a name, a data type, a number of attribute values and the attribute values themselves. All attribute values must be of the same data type - for example, an integer cannot be added to an attribute value consisting of ten characters, or a character value cannot be included in an attribute value consisting of 2 32-bit integers.

Any number of attributes can be assigned to either a vdata or any single field in a vdata. However, each attribute name should be unique within its scope. In other words, a field attribute name must be unique among all attributes of that field, and a vdata attribute name must be unique among all attributes of the vdata it's contained in.

4.8.1 Querying the Index of a Vdata Field Given the Field Name: VSfindex

VSfindex returns the index of specified field given the field name. It returns `SUCCEED` if successful, and `FAIL` otherwise.

TABLE 4I

VSfindex Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
VSfindex (vsffidx)	vdata_id	int32	integer	Vdata identifier.
	field_name	char *	character* (*)	Name of the target vdata field.
	field_index	int32 *	integer	Index of the target vdata field.

4.8.2 Setting the Attribute of a Vdata Field or Vdata: VSsetattr

VSsetattr attaches an attribute to a vdata field or a vdata. If the attribute already exists, the new values will replace the current ones, provided the data type and order have not been changed. If either the data type or the order have been changed, **VSsetattr** will exit on an error condition.

The *field_index* parameter is zero-based, and represents the ordinal location of the field within the vdata - for example, a *field_index* value of `4` would refer to the fifth field in the vdata. If *field_index* is set to `_HDF_VDATA`, the attribute of the specified vdata will be set to the value pointed to by *values*. When using either Fortran-77 version of this routine, specify a *field_index* value of `-1` to attach an attribute to the specified vdata.

VSsetattr returns `SUCCEED` if successful, `FAIL` otherwise.

TABLE 4J

VSsetattr Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
VSsetattr (vsfsnat/vsfscat)	vdata_id	int32	integer	Vdata identifier.
	field_index	int32	integer	Vdata field index.
	attr_name	char *	character* (*)	Name of the attribute.
	data_type	int32	integer	Data type of the attribute.
	count	int32	integer	Number of values the attribute contains.
	values	VOIDP	integer (*)/character* (*)	Buffer containing the attribute values.

4.8.3 Querying the Total Number of Vdata Attributes: VSnattrs

VSnattrs returns the number of attributes of the specified vdata *and* the vdata fields contained in it. This is different from the **VSfnattrs** routine, which returns the number of attributes of a field contained in the specified vdata *or* the specified vdata.

VSnattrs returns total number of attributes assigned to this vdata and its fields when successful, and `FAIL` otherwise

TABLE 4K

VSnattrs Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
VSnattrs (vsfnats)	vdata_id	int32	integer	Vdata identifier.

4.8.4 Querying the Number of Attributes of a Vdata or a Vdata Field: VSfnattrs

VSfnattrs returns the number of attributes attached to the specified vdata field *or* the number of attributes attached to the vdata identified by *vdata_id*. This is different from the **VSnattrs** routine, which returns the number of attributes of the specified vdata *and* the fields contained in it.

If *field_index* is set to `_HDF_VDATA`, the number of attributes attached to the vdata referred to by the value of *vdata_id* is returned. When using the Fortran-77 version of this routine, specify a *field_index* value of `-1` to return the number of vdata attributes.

As with **VSsetattr**, if *field_index* is set to a zero-based integer value, it will be used as the index of the target vdata field, and the attributes attached to that field will be returned.

VSfnattrs returns number of attributes assigned to this vdata or its fields when successful, and `FAIL` otherwise.

TABLE 4L **VSfnattrs Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
VSfnattrs **(vsffnas)**	vdata_id	int32	integer	Vdata identifier.
	field_index	int32	integer	Index of the target field.

4.8.5 Retrieving the Index of a Vdata Attribute Given the Attribute Name: VSfindattr

VSfindattr returns the index of an attribute with the specified name. The attribute must to attached to the vdata identified by the the specified vdata identifier.

If *field_index* is set to `_HDF_VDATA`, the index of the attribute attached to the vdata referred to by the value of *vdata_id* is returned. When using the Fortran-77 version of this routine, specify a *field_index* value of `-1` to return the vdata attribute indices.

As with **VSsetattr**, if *field_index* is set to a zero-based integer value, the value will be used as the index of the target vdata field, and the index of the attribute with the name specifiedby the *attr_name* parameter will be returned.

VSfindattr returns the index of the target attribute if successful, and `FAIL` otherwise.

TABLE 4M **VSfindattr Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
VSfindattr **(vsffdat)**	vdata_id	int32	integer	Vdata identifier.
	field_index	int32	integer	Index of the target field.
	attr_name	char *	character* (*)	Name of the target attribute.

4.8.6 Querying Information on a Given Vdata Attribute: VSattrinfo

VSattrinfo returns the name, data type, number of values, and the size of the values of the specified attribute of the specified vdata field or vdata.

The values of the *attr_name*, *data_type*, *count* and *size* parameters can be set to `NULL`, if the information returned by these parameters are not needed.

The *field_index* in **VSattrinfo** is the same as with the *field_index* parameter in **VSsetattr**. It can be set to either a integer field index, or `_HDF_VDATA` to specify the vdata referred to by *vdata_id*.

VSattrinfo returns `SUCCEED` if successful, and `FAIL` otherwise.

TABLE 4N **VSattrinfo Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
VSattrinfo (vsfainf)	vdata_id	int32	integer	Vdata identifier.
	field_index	int32	integer	Index of the target field
	attr_index	intn	integer	Index of the target attribute.
	attr_name	char *	character* (*)	Returned name of the target attribute.
	data_type	int32 *	integer	Returned data type of the target attribute.
	count	int32 *	integer	Number of values of the target attribute.
	size	int32 *	integer	Size, in bytes, of the values of the target attribute.

4.8.7 Querying the Values of a Given Vdata Attribute: VSgetattr

VSgetattr returns all of the values of the specified attribute of the specified vdata field or vdata.

The *attr_index* parameter is the ordinal, zero-based (as with *field_index* in **VSsetattr** and this routine) index of the target attribute.

If *field_index* is set to `_HDF_VDATA`, the value of the attribute attached to the vdata referred to by the value of *vdata_id* is returned. If *field_index* is set to a zero-based integer value, the value will be used as the index of the target vdata field, and the attribute located at the ordinal position specified by *attr_index* will be returned.

VSgetattr returns `SUCCEED` if successful, `FAIL` otherwise.

TABLE 4O **VSgetattr Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
VSgetattr (vsfgnat/vsfgcat)	vdata_id	int32	integer	Vdata identifier.
	field_index	int32	integer	Index of the target field.
	attr_index	intn	integer	Index of the target attribute.
	values	VOIDP	<valid numeric data type>	Buffer containing attribute values.

4.8.8 Determining if the Given Vdata is an Attribute: VSisattr

As attributes are stored by the HDF library as vdatas, a means of testing whether or not a particular vdata is attribute data is provided by the **VSisattr** routine.

VSisattr returns `TRUE` if the vdata is an attribute, and `FALSE` otherwise.

TABLE 4P

VSisattr Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
VSisattr (vsfisat)	vdata_id	int32	integer	Vdata identifier.

EXAMPLE 7.

Attaching and Querying Vdata Attributes

These examples attach an attribute to the vdata created in Example 3 (in the file named "Example3.hdf") and to the first field of that vdata. Then it checks the number of attributes in the file and reads the attribute data.

C:

```
#include "hdf.h"
#include "vg.h"

#define FILE_NAME "Example3.hdf"
#define VATTR_NAME "Vdata Attribute 1"
#define FATTR_NAME "Field Attribute 1"

main( )
{

int32 file_id, vdata_ref, vdata_id, status;
int32 n_vdattrs, n_fldattrs;
int32 v_type, v_count, v_size, f_type, f_count, f_size;
char vd_attr[6] = {'m','N','p', 'S', 't', '\0'};
char vattr_buf[6];
char vattrname[30], fattrname[30];
float32 fld_attr[2] = {30.452, 5.467};
float32 fattr_buf[2];

/* Open the HDF file. */
file_id = Hopen(FILE_NAME, DFACC_RDWR, 0);

/* Initialize the vdata interface. */
status = Vstart(file_id);

/* Get the reference number of the target vdata. */
vdata_ref = VSfind(file_id, "Example Vdata");

/* Attach to the target vdata. */
vdata_id = VSattach(file_id, vdata_ref, "w");

/* Attach an attribute to the vdata. */
status = VSsetattr(vdata_id, _HDF_VDATA, VATTR_NAME, DFNT_CHAR,\
                6, vd_attr);

/* Attach an attribute to the first field in the vdata. */
```

```
status = VSsetattr(vdata_id, 0, FATTR_NAME, DFNT_FLOAT32, 2,
            fld_attr);

/* Get the number of attributes attached to the first field -
should be 1. */
n_fldattrs = VSfnattrs(vdata_id, 0);

/* Get the total number of field and vdata attributes.
This should be 2. */
n_vdattrs = VSnattrs(vdata_id);

/* Get information about the vdata attribute. */
status = VSattrinfo(vdata_id, _HDF_VDATA, 0, vattrname, \
        &v_type, &v_count, &v_size);

/* Get information about the field attribute. */
status = VSattrinfo(vdata_id, 0, 0, fattrname, &f_type,
            &f_count, &f_size);

/* Get the vdata attribute. */
status = VSgetattr(vdata_id, _HDF_VDATA, 0, vattr_buf);

/* Get the field attribute. */
status = VSgetattr(vdata_id, 0, 0, fattr_buf);

/* Detach from the vdata, close the VS interface and the file. */
status = VSdetach(vdata_id);
status = Vend(file_id);
status = Hclose(file_id);

}
```

FORTRAN:

```
     PROGRAM CREATE QUERY ATTRS

      integer file_id, vdata_ref, vdata_id, status
      integer n_fldattrs, n_vdattrs
      integer hopen, vfstart, vsffnd, vsfatch, vsfdtch, hclose
      integer vfend, vsfscat, vsfsnat, vsffnas, vsfnats
      integer vsfainf, vsfgnat, vsfgcat
      integer v_type, v_count, v_size, f_type, f_count, f_size
      double precision fld_attr(2), fattr_buf(2)
      character*20 vattrname, vattr_buf
      character*50 filename, vd_attr, fattr_name
      parameter (filename = 'Example3.hdf',
     +      vattr_name = 'Vdata Attribute 1',
     +      vd_attr = 'mnpst',
     +      fattr_name = 'Field Attribute 1'
     +     )

C     The following parameters are defined in hdf.inc.
      integer DFACC_RDWR, DFNT_CHAR, DFNT_FLOAT32, _HDF_VDATA
      parameter (DFACC_RDWR = 3, DFNT_CHAR = 4, DFNT_FLOAT32 = 5,
     +           _HDF_VDATA = -1)

      fld_attr(1) = 30.452
      fld_attr(2) = 5.467

C     Open an HDF file with full access.
      file_id = hopen(filename, DFACC_RDWR, 0)

C     Initialize the VS interface.
```

```
      status = vfstart(file_id)

C     Get the reference number of the target vdata.
      vdata_ref = vsffnd(file_id, 'Example Vdata')

C     Attach to the target vdata.
      vdata_id = vsfatch(file_id, vdata_ref, 'w')

C     Attach an attribute to the vdata.
      status = vsfscat(vdata_id, _HDF_VDATA, vattr_name,
     +                 DFNT_CHAR, 5, vd_attr)

C     Attach an attribute to the first field of the vdata.
      status = vsfsnat(vdata_id, _HDF_VDATA, fattr_name,
     +                 DFNT_FLOAT32, 2, fld_attr)

C     Get the number of attributes attached to the first field.
      n_fldattrs = vsffnas(vdata_id, 0)

C     Get the total number of field and vdata attributes.
      n_vdattrs = vsfnats(vdata_id)

C     Get information about the vdata attribute.
      status = vsfainf(vdata_id, _HDF_VDATA, 0, vattrname,
     +               v_type, v_count, v_size)

C     Get information about the field attribute.
      status = vsfainf(vdata_id, 0, 0, vattrname, f_type,
     +               f_count, f_size)

C     Get the vdata attribute.
      status = vsfgnat(vdata_id, _HDF_VDATA, 0, vattr_buf)

C     Get the field attribute.
      status = vsfgcat(vdata_id, 0, 0, fattr_buf)

C     Detach from the vdata, close the VS interface and the file.
      status = vsfdtch(vdata_id)
      status = vfend(file_id)
      status = hclose(file_id)

      end
```

4.9 Obtaining Information About a Specific Vdata

Once a vdata has been located, its contents must be determined. In this section three categories of routines that obtain vdata information are described:

- A general inquiry routine named **VSinquire**.
- As a supplement to **VSinquire**, a set of ***vdata query*** routines with names prefaced by "VSQuery".
- A set of routines prefaced by "VS". Some of the functionality of these routines are duplicated in the functionality of **VSinquire**.
- A set of ***field query*** routines with names prefaced by "VF".

4.9.1 Obtaining Vdata Information: VSinquire

VSinquire is a general inquiry routine used to request information about the contents of a vdata. It has the following syntax:

```
status = VSinquire(vdata_id, n_records, interlace, fields, vdata_size,
                   vdata_name);
```

The `n_records` parameter contains the returned number of records in the target vdata, the `interlace` parameter contains the returned interlace mode of the vdata contents, the `fields` parameter is a comma-separated list of the returned names of all fields in the vdata, the `vdata_size` parameter is the returned size, in bytes, of the vdata and the `vdata_name` parameter contains the returned name of the vdata.

If any of the parameters are set to `NULL`, the corresponding data is not returned.

The parameters of **VSinquire** are further defined in Table 4Q below.

TABLE 4Q **VSinquire Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
VSinquire (vsfinq)	vdata_id	int32	integer	Vdata identifier.
	n_records	int32 *	integer	Number of records in the vdata.
	interlace	int32 *	integer	Interlace mode.
	fieldname_list	char *	character* (*)	Buffer for the returned list of field names.
	vdata_size	int32 *	integer	Size in bytes of a vdata record.
	vdata_name	char *	character* (*)	Name of the vdata.

EXAMPLE 8. **Obtaining Vdata Information**

The following examples search for the vdata named "Example Vdata".

C:

```
#include "hdf.h"

#define FILE_NAME "Example3.hdf"

main( )
{

int32 file_id, vdata_ref, vdata_id, status;
int32 n_records, interlace, vdata_size;
char fields[30], vdata_name[VSNAMELENMAX];
int8 found_fields;

/* Open the HDF file. */
file_id = Hopen(FILE_NAME, DFACC_RDONLY, 0);

/* Initialize the vdata interface. */
status = Vstart(file_id);

vdata_ref = -1;
found_fields = 0;

/* Attach to each vdata and search for the fields. */
```

```
while ((vdata_ref = VSgetid(file_id, vdata_ref)) != -1) {
   vdata_id = VSattach(file_id, vdata_ref, "r");
    status = VSinquire(vdata_id, &n_records, &interlace, fields,
                           &vdata_size, vdata_name);
    status = VSdetach(vdata_id);
   if (!strncmp(vdata_name, "Example Vdata", 20)) {
      found_fields = 1;
      break;
   } else VSdetach(vdata_id);
}

if (!found_fields)
   printf("Vdata name - Example Vdata - was not found.\n");
else
   printf("Vdata name - Example Vdata - found, vdata id %d.\n",
                         vdata_ref);

/* Detach from the vdata, close the VS interface and the file. */
status = VSdetach(vdata_id);
status = Vend(file_id);
status = Hclose(file_id);

}
```

FORTRAN:

```
     PROGRAM SEARCH VDATANAME

      integer*4 file_id, vdata_ref
      integer*4 n_records, interlace, vdata_size, vdata_id
      character fields *30, vdata_name *20
      logical found_fields
      integer hopen, vsfgid, vsfatch, vsfinq, vsfdtch, hclose

C     DFACC_RDONLY is defined in hdf.inc.
      integer DFACC_RDONLY
      parameter (DFACC_RDONLY = 1)

      character target_vdata_name *20
      parameter (target_vdata_name = 'Example Vdata')

C     Open an HDF file with full access.
      file_id = hopen('Example3.hdf', DFACC_RDONLY, 0)

C     Initialize the VS interface.
      call vfstart(file_id)

      vdata_ref = -1
      found_fields = .FALSE.

C     Attach to each vdata and search for the fields.
10    vdata_ref = vsfgid(file_id, vdata_ref)
      if (vdata_ref .ne. -1) then
        vdata_id = vsfatch(file_id, vdata_ref, 'r')
        status = vsfinq(vdata_id, n_records, interlace, fields,
     +                 vdata_size, vdata_name)
        status = vsfdtch(vdata_id)

        if (vdata_name .eq. target_vdata_name) then
           found_fields = .TRUE.
           go to 20
        else
            go to 10
```

```
          end if
        end if

20      if (found_fields .eq. .FALSE.) then
          print *, 'Vdata name - Example Vdata - was not found'
        else
          print *, 'Vdata name - Example Vdata - found, vdata id ',
     +               vdata_ref
        end if

C       Detach from the vdata, close the Vset interface and the file.
        status = vfend(file_id)
        status = hclose(file_id)

        end
```

4.9.2 VSQuery Vdata Information Retrieval Routines

The syntax of the VSQuery routines are as follows:

```
status = VSQueryname(vdata_id, *vdata_name);
status = VSQueryclass(vdata_id, *vdata_class);
status = VSQueryfields(vdata_id, fields);
status = VSQueryinterlace(vdata_id, *interlace);
status = VSQuerycount(vdata_id, *n_records);
vdata_tag = VSQuerytag(vdata_id);
vdata_ref = VSQueryref(vdata_id);
status = VSQueryvsize(vdata_id, *vdata_vsize);
```

All VSQuery routines except **VSQuerytag** and **VSQueryref** have two arguments. The second argument is the type of vdata information being requested. **VSQuerytag** and **VSQueryref** are functions that retrieve the vdata information through their return value.

- **VSQueryname** retrieves the name of the specified vdata.
- **VSQueryfields** retrieves the name of the fields in the specified vdata.
- **VSQueryinterlace** retrieves the interlace mode of the specified vdata.
- **VSQuerytag** returns the tag of the specified vdata.
- **VSQuerycount** returns the number of records in the specified vdata.
- **VSQueryref** returns the reference number of the specified vdata
- **VSQueryvsize** retrieves the size, in bytes, of a record in the specified vdata.

The parameters of these routines are listed in the following table.

TABLE 4R **VSQuery Routine Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
VSQueryname (vsqfname)	vdata_id	int32	integer	Vdata identifier.
	vdata_name	char *	character* (*)	Name of the vdata.
VSQueryfields (vsqflds)	vdata_id	int32	integer	Vdata identifier.
	fields	char *	character* (*)	Comma-separated list of the field names in the vdata.
VSQueryinterlace (vsqfintr)	vdata_id	int32	integer	Vdata identifier.
	interlace	int32 *	integer	Interlace mode.

VSQuerycount (vsqnfelt)	vdata_id	int32	integer	Vdata identifier.
	n_records	int32 *	integer	Number of records in the vdata.
VSQueryvsize (vsqfvsiz)	vdata_id	int32	integer	Vdata identifier.
	vdata_size	int32 *	integer	Size in bytes of the vdata record.
VSQuerytag (vsqtag)	vdata_id	int32	integer	Vdata identifier.
VSQueryref (vsqref)	vdata_id	int32	integer	Vdata identifier.

4.9.3 VS Vdata Information Retrieval Routines

Some routines with names prefaced by "VS" are used to obtain specific types of vdata information. Here is the syntax of these routines:

```
num_of_elements = VSelts(vdata_id);
status = VSgetclass(vdata_id, vdata_class);
num_of_fields = VSgetfields(vdata_id, field_names);
interlace_mode = VSgetinterlace(vdata_id);
status = VSgetname(vdata_id, vdata_name);
size_of_fields = VSsizeof(vdata_id, field_name_list);
```

With the exception of **VSgetclass**, the information obtainable through these routines can also be obtained through **VSinquire**. **VSinquire** provides a way to query commonly used vdata information with one standardized routine call. However, as there are situations where the HDF programmer may not want all of the information **VSinquire** provides, the VS routines have also been provided.

- **VSelts** returns the number of elements in the specified vdata.
- **VSgetclass** retrieves the class of the specified vdata.
- **VSgetfields** returns the number of and retrieves the names of all fields in the specified vdata.
- **VSgetinterlace** retrieves the interlace mode of the specified vdata.
- **VSgetname** retrieves the name of the specified vdata.
- **VSsizeof** returns the size, in bytes, of the specified field or fields.

The parameters of these routines are described in the following table.

TABLE 4S

VSelts, VSgetclass, VSgetfields, VSgetinterlace, VSgetname and VSsizeof Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran -77	
VSelts (vsfelts)	vdata_id	int32	integer	Vdata identifier.
VSgetclass (vsfcls)	vdata_id	int32	integer	Vdata identifier.
	vdata_class	char *	character* (*)	Pointer to the class name.
VSgetfields (vsfgfld)	vdata_id	int32	integer	Vdata identifier.
	fields	char *	character* (*)	Pointer to the field names.
VSgetinterlace (vsfgint)	vdata_id	int32	integer	Vdata identifier.
VSgetname (vsfgnam)	vdata_id	int32	integer	Vdata identifier
	vdata_name	char *	character* (*)	Pointer to the vdata name.
VSsizeof (vsfsiz)	vdata_id	int32	integer	Vdata identifier.
	fields	char *	character* (*)	List of field names to be queried.

4.9.4 VF Field Information Retrieval Routines

Routines whose names of prefaced by “VF” are used for obtaining information about specific fields in a vdata. Here is the syntax of these routines:

```
field_name = VFfieldname(vdata_id, field_index);
field_file_size = VFfieldesize(vdata_id, field_index);
field_mem_size = VFfieldisize(vdata_id, field_index);
num_of_fields = VFnfields(vdata_id);
field_order = VFfieldorder(vdata_id, field_index);
field_type = VFfieldtype(vdata_id, field_index);
```

The second, `field_index` is the index number that uniquely identifies the location of the field within the vdata. Field index numbers are assigned in increasing order and are zero-based.

The values returned by each of the VF routines are as follows:

- **VFfieldname** returns the name of the specified field.
- **VFfieldesize** returns the size of the specified field as stored in the HDF file. This is the size of the field as tracked by the HDF library.
- **VFfieldisize** returns the size of the specified field as stored in memory. This is the native machine size of the field.
- **VFnfields** returns the number of fields in the specified vdata.
- **VFfieldorder** returns the order of the specified field.
- **VFfieldtype** returns the data type of the specified field.

If unsuccessful, these routines return `FAIL` (or `-1`). The parameters of these routines are described in the following table.

TABLE 4T **VFfieldname, VFfieldesize, VFfieldisize, VFnfields, VFfieldorder and VFfieldtype Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
VFfieldname (vffname)	vdata_id	int32	integer*4	Vdata identifier.
	field_index	int32	integer	Field index.
VFfieldesize (vffesiz)	vdata_id	int32	integer	Vdata identifier.
	field_index	int32	integer	Field index.
VFfieldisize (vffisiz)	vdata_id	int32	integer	Vdata identifier.
	field_index	int32	integer	Field index.
VFnfields (vfnflds)	vdata_id	int32	integer	Vdata identifier.
VFfieldorder (vffordr)	vdata_id	int32	integer	Vdata identifier.
	field_index	int32	integer	Field index.
VFfieldtype (vfftype)	vdata_id	int32	integer	Vdata identifier.
	field_index	int32	integer	Field index.

Chapter 5

Vgroups (V API)

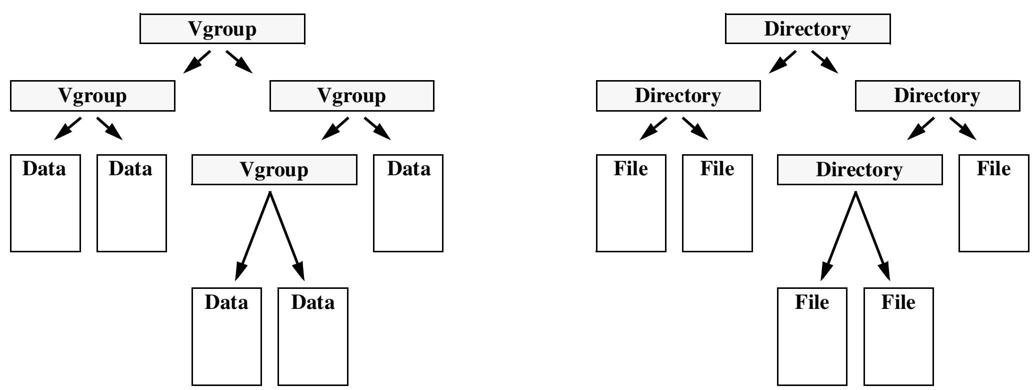
5.1 Chapter Overview

This chapter describes the vgroup data model and the vgroup API. The first section describes the vgroup data model. The second section presents the vgroup API, followed by a presentation of a programming model for vgroups. The next three sections describe the use of the API in accessing vgroups, creating vgroups and reading from vgroups. The final two sections cover vgroup utilities and obsolete vgroup routines.

5.2 The Vgroup Data Model

A ***vgroup*** is a structure designed to associate related objects. Data organization within a vgroup resembles the UNIX file system. (See Figure 5a.) The general structure of vgroups are similar to UNIX directories or subdirectories in that a vgroup may contain references to other vgroups or data objects. In previous versions of HDF, the data objects in a vgroup were limited to vdatas. Any HDF data object can now be included within a vgroup.

FIGURE 5a **Similarity of the HDF Vgroup Structure and the UNIX File System**



5.2.1 Vgroup Names and Classes

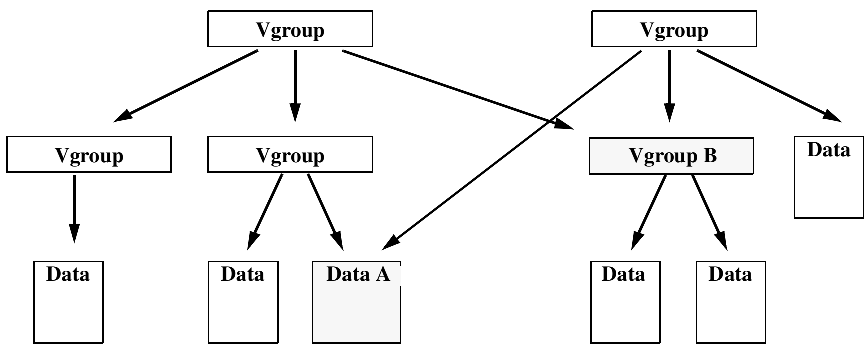
Vgroups have attributes that can be used for the purpose of searching for and classifying them. Two such attributes are the vgroup name and the vgroup class. A ***vgroup name*** is a character string that can be used to describe the contents of a specific vgroup. A vgroup name can be used as a primary key in searches.

A ***vgroup class*** is also a character string and can be also used in searches for a specific vgroup. The purpose of a vgroup class is to allow an application to determine the intended use of its members. For example, a vdata object named "Storm Tracking Data - 5/11/94" and another vdata object named "Storm Tracking Data - 6/23/94" can be grouped together under a vgroup named "Storm Tracking Data - 1994". If the data was collected in Anchorage, Alaska the class name might be "Anchorage Data", particularly if other vgroups contain storm track data collected in different locations. The specific use of the vgroup name and class name is solely determined by the HDF users.

5.2.2 Vgroup Organization

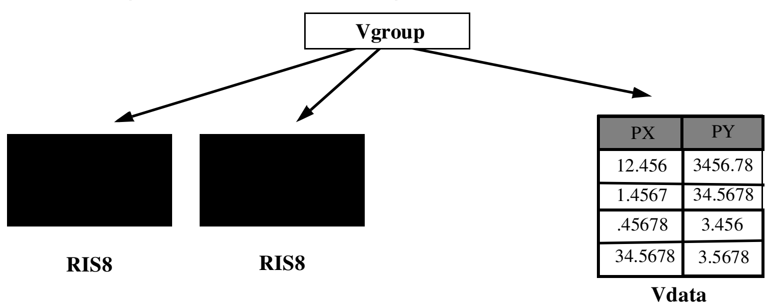
There are many ways to organize vgroups through the use of the Vgroup API. Vgroups may contain any number of groups and data objects, including data objects and vgroups that are members of other vgroups. Therefore, an object may have more than one parent vgroup. For example, Data object A and Vgroup B, shown in Figure 5b, are members of multiple vgroups with different organizational structures.

FIGURE 5b **Sharing Vgroups and Vdatas Among Vgroups**



A vgroup can contain any combination of HDF data objects. Figure 5c illustrates a vgroup that contains two raster image sets and a vdata.

FIGURE 5c **A Vgroup Containing Two 8-Bit Raster Image Sets, or RIS8 Objects, and a Vdata**



5.2.3 Vgroup Conventions: An Example Using Vsets

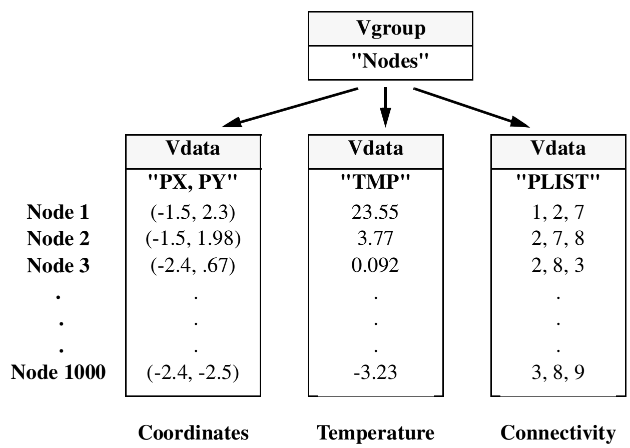
Although vgroups can contain any combination of objects, it is often useful to establish conventions on the content and structure of vgroups. For example, a type of vgroup called a ***vertex set*** or ***vset***, is often used by scientific and graphics programmers to describe the surfaces of an object as well as its properties. This structure is used by NCSA Polyview for viewing meshes and other objects that can be described by polygonal data.

A vset consists of one list of coordinate data, one list of connectivity data and one list of node property data. These three lists are stored in separate vdata objects within the vset.

Each n-dimensional coordinate in the coordinate list defines the relative location of a vertex, or ***node***. Each item of connectivity data represents an edge between two vertices and a list of this edge data is referred to as a ***connectivity list*** , which describes one polygon. Connectivity lists are a table of comma-separated numbers, with each number representing the position of one particular node in the table of nodes. For example, the number "2" as an item in a connectivity list would represent the second entry in the node table. ***Node properties*** are user-defined values attached to each node within the polygon and can be numbers or characters.

For example, consider a heated mesh of 400 triangles formed by connecting 1000 nodes. A vset describing this mesh might contain the coordinates of the vertices, the temperature value of the vertices and a connectivity list describing the edges of the triangles. This vset configuration is the one used most often by graphics applications like NCSA Polyview and is illustrated in Figure 5d below.

FIGURE 5d **Vset Structure Describing a Heated Mesh**



1.3 The Vgroup API

The Vgroup interface consists of routines for accessing, creating and getting information about vgroups.

1.3.1 Vgroup Library Routines

Vgroup API routine names are prefaced by "V". These routines are categorized as follows:

- ***Access routines*** open files, initialize the Vgroup interface and access individual vgroups. They also terminate access to vgroups and the Vgroup interface and close HDF files.
- ***Create routines*** organize, label and add data objects to vgroups.
- ***File inquiry routines*** return information about how vgroups are stored in a file. They are useful for locating vgroups in a file.
- ***Vgroup inquiry routines*** provide specific information about a specific vgroup. This information includes the class, name, member count and additional member information.

The Vgroup API routines, along with the two H interface routines used to open and close HDF files for vgroup access, are listed in Table 5A below.

TABLE 5A **Vgroup Interface Routines**

Category	Routine Name		Description
	C	Fortran-77	
Access	Vstart	vfstart	Initializes the V interface.
	Vattach	vfatch	Establishes access to a vgroup.
	Vdetach	vsfdtch	Terminates access to a vgroup.
	Vend	vfend	Terminates access to the V interface.
Create	Vsetclass	vfscls	Assigns a class to a vgroup.
	Vsetname	vfsnam	Assigns a name to a vgroup.
	Vinsert	vfinsrt	Adds a vgroup or vdata to an existing vgroup.
	Vaddtagref	vfadtr	Adds any HDF data object to an existing vgroup.
	Vsetattr	vfsnatt/vfscatt	Sets the attribute of a vgroup.
File Inquiry	Vlone	vflone	Returns the reference numbers of vgroups not included in other vgroups.
	Vgetid	vfgid	Returns the reference number for the next vgroup in the HDF file.
Vgroup Inquiry	Vinquire	vfinq	Returns general information about a vgroup.
	Vgetclass	vfgcls	Returns the class of the specified vgroup.
	Vgetname	vfgnam	Returns the name of the specified vgroup.
	Visvg	vfisvg	Checks if a vgroup identifier belongs to a vgroup within a vgroup.
	Visvs	vfisvs	Checks if a vdata identifier belongs to a vdata within a vgroup.
	Vgettagref	vfgttr	Retrieves a tag/ reference number pair for a data object in the specified vgroup.
	Vntagrefs	vfntr	Returns the number of tag/reference number pairs contained in the specified vgroup.
	Vgetnext	vfgnxt	Returns the identifier of the next vgroup or vdata in a vgroup.
	Vgettagrefs	vfgttrs	Retrieves the tag/reference pairs of all of the data objects with a vgroup.
	Vinqtagref	vfinqtr	Checks if an object belongs to a vgroup.
	Vgetversion	None	Queries the vgroup version of a given vgroup.
	Vnaattrs	vfnatts	Queries the total number of vgroup attributes.
	Vfindattr	vffdatt	Retrieves the index of an attribute given the attribute name.
	Vattrinfo	vfainfo	Queries information on a given vgroup attribute.
	Vgetattr	vfgnatt/vfgcatt	Queries the values of a given attribute.

1.3.2 Vgroup Identifiers in the Vgroup Interface

The Vgroup interface identifies vgroups in several ways. Before a vgroup is attached, it is uniquely identified by its ***name***, ***class*** and ***reference number***. When a vgroup is attached, it is assigned a vgroup identifier or ***vgroup id***. After a vgroup has been attached, its vgroup id is used by the Vgroup interface routines to access vgroups.

The name, class, and reference number are values assigned to the vgroup when it is created. The name and class are normally assigned by the calling program and may be changed throughout the scope of the vgroup object. However, the reference number is assigned by the HDF library and not changed.

1.4 Programming Model for the Vgroup Interface

The programming model for accessing vgroups is as follows:

1. Open a file.
2. Initialize the V interface
3. Open a vgroup.
4. Perform the desired operations on the vgroup.
5. Dispose of the vgroup id.
6. Terminate access to the V interface.
7. Close the file.

To access a single vgroup in an HDF file, the calling program must contain the following calls:

C:

```
file_id = Hopen(filename, file_access_mode, n_dds);
status = Vstart(file_id);
vgroup_id = Vattach(file_id, vgroup_ref, vgroup_access_mode);
<Optional operations>
status = Vdetach(vgroup_id);
status = Vend(file_id);
status = Hclose(file_id);
```

FORTRAN:

```
file_id = hopen(filename, file_access_mode, n_dds)
status = vfstart(file_id)
vgroup_id = vsfatch(file_id, vdata_ref, vgroup_access_mode)
<Optional operations>
status = vfdtch(vgroup_id)
status = vfend(file_id)
status = hclose(file_id)
```

The calling program must obtain a separate vgroup id for each vgroup to be accessed.

1.4.1 Accessing Files and Vgroups: Hopen, Vstart and Vattach

An HDF file must be opened by **Hopen** before it can be accessed using the V interface. **Hopen** is described in Chapter 2, titled *HDF Fundamentals*.

The Vgroup interface routines are used in a similar manner to the V interface routines. Before performing operations on a vgroup, a calling program must call **Vstart** for every file it intends to access and **Vattach** for every vgroup it intends to access. **Vstart** initializes the internal vgroup structures in a file. Consequently, any HDF file involved in a vgroup operation must be initialized by **Vstart** before the operation begins. **Vstart** takes one argument, the file id returned by **Hopen**.

Vattach provides access to an individual vgroup for all read and write operations. It takes three arguments: `file_id`, `vgroup_ref` and the `access_mode`. The argument `file_id` is a file identifier returned by **Hopen** and `vgroup_ref` is the reference number that identifies the vgroup to be accessed. Specifying a nonexistent reference number will return an error, specifying a value of -1 for the reference number will create a new vgroup and specifying an existing reference number will provide access to the corresponding vgroup. To find the reference number of a vgroup, consider using either **Vgetnext** or **Vgetid**.

The `access_mode` parameter in **Vattach** specifies the type of access ("r" or "w") required for operations on the selected vgroup. This argument is not currently used; all vgroups are automatically opened with read and write access. Multiple attaches may be made to the same vgroup, which will result in several vgroup identifiers being assigned to the same vgroup.

The parameters of **Vstart** and **Vattach** are defined in Table 5B below.

1.4.2 Terminating Access to Vgroups and Files: Vdetach, Vend and Hclose

Successfully terminating access to a vgroup requires one **Vdetach** call for every **Vattach** call made. The calling program must also call **Vend** once for every **Vstart** call made. All **Vdetach** calls must occur prior to the first **Vend** call for the same file. **Vdetach** terminates access by updating internal library structures and frees all memory associated with the vgroup. Once a vgroup is detached, its vgroup id is invalid and any attempts to access it will result in an error condition.

Vend releases all internal data structures allocated by calling **Vstart**. Its one argument is `file_id`, the file identifier returned by **Hopen**. **Vend** must be called once per file and only after all vgroups are detached. Attempts to use the Vgroup interface after calling **Vend** will produce errors.

Hclose terminates access to a file only if there are no vgroup ids attached to the file. To successfully close a file, individually release all vgroup idsby making one **Vdetach** call for each **Vattach** call.

The parameters of **Vdetach** and **Vend** are also defined in the following table.

TABLE 5B **Vstart, Vattach, Vdetach and Vend Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
Vstart (**vstart**)	file_id	int32	integer	File identifier.
Vattach (**vfatch**)	file_id	int32	integer	File identifier.
	vgroup_ref	int32	integer	Reference number for existing vgroup or -1 for new.
	access_mode	char *	character* (*)	Access mode of the vgroup operation.
Vdetach (**vfdtch**)	vgroup_id	int32	integer	Vgroup identifier.
Vend (**vend**)	file_id	int32	integer	IFile identifier.

1.5 Creating and Writing to a Vgroup

There are two steps involved in the creation of vgroups. First, the vgroup must be created, then the data objects must be inserted into it. Any HDF data object can be inserted into a vgroup. Creation and insertion operations are usually performed at the same time - although, as with the VS interface routines it isn't required to do so.

1.5.1 Creating a Vgroup:Vattach

Creating a vgroup involves the following steps:

1. Open the file.
2. Initialize the V interface.
3. Create the new vgroup.
4. Optionally assign a vgroup name.
5. Optionally assign a vgroup class.
6. Optionally insert the data objects.

7. Terminate access to the vgroup.
8. Terminate access to the V interface.
9. Close the file.

These steps correspond to the following calling sequence:

C:
```
file_id = Hopen(filename, file_access_mode, n_dds);
status = Vstart(file_id);
vgroup_id = Vattach(file_id, vgroup_ref, vgroup_access_mode);
status = Vsetname(vgroup_id, vgroup_name);
status = Vsetclass(vgroup_id, vgroup_class);
num_of_tag_refs = Vaddtagref(vgroup_id, tag, ref); or
obj_pos = Vinsert(vgroup_id, v_id);
status = Vdetach(vgroup_id);
status = Vend(file_id);
status = Hclose(file_id);
```

FORTRAN:
```
file_id = hopen(filename, file_access_mode, n_dds)
status = vfstart(file_id)
vgroup_id = vfatch(file_id, vgroup_ref, vgroup_access_mode)
status = vfsnam(vgroup_id, vdata_name)
status = vfscls(vgroup_id, vdata_class)
num_of_tag_refs = vfadtr(vgroup_id, tag, ref) or
obj_pos = vfinsrt(vgroup_id, v_id)
status = vfdtch(vgroup_id)
status = vfend(file_id)
status = hclose(file_id)
```

In the routines that follow, `vgroup_id` is the vgroup identifier returned by **Vattach.**

When a new vgroup is created, the value of `vgroup_ref` must be set to `-1` and the value of `access_mode` must be `"w"`.

1.5.2 Assigning a Vgroup Name and Class: Vsetname and Vsetclass

Vsetname assigns a name to a vgroup. The parameter `vgroup_name` is a character string with the name to be assigned to the vgroup. If **Vsetname** is not called, a vgroup name is set to a zero-length string. A name may be assigned and reset any time after the vgroup is created.

Vsetclass assigns a class to a vgroup. The parameter `vgroup_class` is a character string with the class name to be assigned to the vgroup. If **Vsetclass** is not called, a vgroup class is set to a zero-length string. As with the vgroup names, the class may be set and reset at any time after the vgroup is created.

Vsetname and **Vsetclass** are further described below. (See Table 5C on page 148.)

1.5.3 Inserting Any HDF Data Object Into a Vgroup: Vaddtagref

The routine most commonly used for inserting objects into a vgroup is **Vaddtagref**. Data objects may be added to a vgroup when the vgroup is created or at any point thereafter.

The `tag` and `ref` parameters in **Vaddtagref** are the tag and reference number of the data object to be inserted into the vgroup respectively. **Vaddtagref** returns the total number of tag and reference number pairs in the vgroup if the operation is successful and `FAIL` (or `-1`) otherwise.

Vaddtagref is firther described below. (See Table 5C on page 148.)

1.5.4 Inserting a Vdata or Vgroup Into a Vgroup: Vinsert

Vinsert is a routine designed specifically for inserting vdatas or vgroups into a parent vgroup. To use **Vinsert**, you must provide the vdata id of the parent vgroup, as well as the vdata id or the vgroup id of the vdata or vgroup to be inserted. These ids are obtained by inserting the vgroups and vdatas in the manner described in the preceeding subsection and in Chapter 4, titled *Vdatas (VS API)*.

The `v_id` parameter in **Vinsert** is either a vdata id or a vgroup id, depending on whether a vdata or vgroup is to be inserted. **Vinsert** returns the index of the inserted vdata or vgroup if the operation is successful and `FAIL` (or `-1`) otherwise.

Vinsert is further defined in the following table.

TABLE 5C

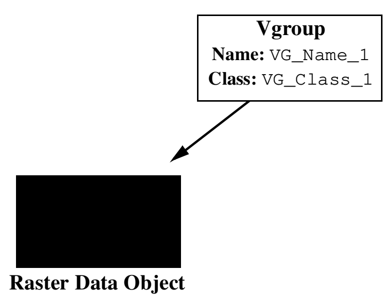
Vsetname, Vsetclass, Vaddtagref and Vinsert Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
Vsetname (**vfsnam**)	vgroup_id	int32	integer	Vgroup identifier.
	vgroup_name	char *	character* (*)	Vgroup name.
Vsetclass (**vfscls**)	vgroup_id	int32	integer	Vgroup identifier.
	vgroup_class	char *	character* (*)	Vgroup class.
Vaddtagref (**vfadtr**)	vgroup_id	int32	integer	Vgroup identifier.
	tag	int32	integer	Tag of the object to be inserted.
	ref	int32	integer	Reference number of the object to be inserted.
Vinsert (**vfinsrt**)	vgroup_id	int32	integer	Vgroup identifier.
	v_id	int32	integer	Vgroup or vdata identifier of the object to be inserted.

EXAMPLE 1.

Creating a Vgroup Containing a Raster Image

The following examples use **Vaddtagref** to insert an 8-bit raster image into a vgroup that is created with the name "VG_Name_1" and the class "VG_Class_1". The following figure illustrates what these examples do. The C and Fortran-77 code uses two different means of initializing the example raster image set, but in most real-world applications this set would be read from an external data file using the RIS8 routines described in Chapter 6, titled *8-Bit Raster Images (DFR8 API)*.



C:

```
#include "hdf.h"

#define HEIGHT 6
```

```
#define WIDTH 5

main( )
{

    int32 file_id, vgroup_id, vdata_id, status;
    uint16 tag, ref;

    /* Construct the image to be written to the vgroup. */
    static uint8 raster_data[HEIGHT][WIDTH] =
                           { 1, 2, 3, 4,  5,
               6,  7,  8,  9, 10,
               11, 12, 13, 14, 15,
               16, 17, 18, 19, 20,
               21, 22, 23, 24, 25,
               26, 27, 28, 29, 30 };

    /* Open an HDF file with full access. */
    file_id = Hopen("Example1.hdf", DFACC_CREATE, 0);

    /* Initialize HDF for subsequent vgroup/vdata access. */
    status = Vstart(file_id);

    /* Create a vgroup. */
    vgroup_id = Vattach(file_id, -1, "w");

    /* Set the name and class for this vgroup. */
    status = Vsetname(vgroup_id, "VG_Name_1");
    status = Vsetclass(vgroup_id, "VG_Class_1");

    /* Write the data to file and determine its tag and ref number. */
    status = DFR8addimage("Example1.hdf", (VOIDP) raster_data, WIDTH,
                          HEIGHT, 0);
    ref = DFR8lastref( );

    /* This tag definition is from hdf.h. */
    tag = DFTAG_RI8;

    /* Insert the data image into the vgroup. */
    status = Vaddtagref(vgroup_id, tag, ref);

    /* Terminate access to the vgroup interface. */
    status = Vdetach(vgroup_id);
    status = Vend(file_id);

    /* Close the HDF file. */
    status = Hclose(file_id);

}
```

FORTRAN:

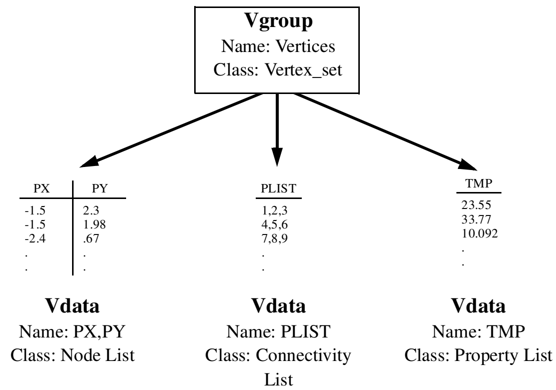
```
      PROGRAM VGROUP INSERT

       integer file_id, vgroup_id, status, tag, ref
       integer*4 raster_data(6, 5)
       integer i, j, k
       integer hopen, vfatch, vfsnam, vfscls
       integer d8aimg, d8lref, vfadtr, vfdtch, vfatch
       integer hclose, vfstart, vfend
       integer DFACC_CREAT
       parameter (DFACC_CREAT = 4)
```

```
C     Construct the data image to be written to the vgroup.
      do 20 j = 1, 5
        do 10 i = 1, 6
            raster_data(i, j) = k
            k = k + 1
10      continue
20    continue

C     Open an HDF file with full access.
C     DFACC_CREAT is defined in hdf.inc.
      file_id = hopen('Example1.hdf', DFACC_CREAT, 0)

C     Open a vgroup with write access.
      status = vfstart(file_id)
      vgroup_id = vfatch(file_id, -1, 'w')

C     Set the name and class for this vgroup.
      status = vfsnam(vgroup_id, 'VG_Name_1')
      status = vfscls(vgroup_id, 'VG_Class_1')

C     Write the data to file and determine its tag and reference
C     number.
      status = d8aimg('Example1.hdf', raster_data, 6, 5, 0)
      ref = d8lref( )

C     The number '202' corresponds to the value of DFTAG_RI8 in
C     hdf.inc.
      tag = 202

C     Insert the data image into the vgroup.
      status = vfadtr(vgroup_id, tag, ref)

C     Terminate access to the V interface.
      status = vfdtch(vgroup_id)
      status = vfend(file_id)

C     Close the HDF file.
      status = hclose(file_id)

      end
```

EXAMPLE 2.

Inserting Three Vdatas Into a Vgroup to Create a Vset

Consider a heated mesh of twenty triangles formed by connecting thirty nodes. The following examples demonstrate how to create a vset containing three vdatas where the vdatas are the coordinate position of each node, the connectivity list describing each relation of three vertices (or triangles) in the vset and a temperature value for each node. The following figure illustrates what is done in the examples. In real-world applications, the contents of the vdatas would be read in from external files, not initialized as in these examples.



C:

```
#include "hdf.h"

main( )
{

int32 file_id, vgroup_id, vdata_id, status;
int32 num_of_elements, vg_position;
float32 pxy[30][2] = {-1.5, 2.3, -1.5, 1.98, -2.4, .67,
                -3.4, 1.46, -.65, 3.1, -.62, 1.23,
                -.4, 3.8, -3.55, 2.3, -1.43, 2.44,
                .23, 1.13, -1.4, 5.43, -1.4, 5.8,
                -3.4, 3.85, -.55, .3, -.21, 1.22,
                -1.44, 1.9, -1.4, 2.8, .94, 1.78,
                -.4, 2.32, -.87, 1.99, -.54, 4.11,
                -1.5, 1.35, -1.4, 2.21, -.22, 1.8,
                -1.1, 4.55, -.44, .54, -1.11, 3.93,
                -.76, 1.9, -2.34, 1.7, -2.2, 1.21};

float32 temp[30];
uint8 mesh[20][3];
uint8 i, j, k = 0;

/* Open an HDF file with full access. */
file_id = Hopen("Example2.hdf", DFACC_CREATE, 0);

/* Initialize HDF for subsequent vgroup/vdata access. */
status = Vstart(file_id);

/* Initialize the data buffer arrays. */
for (i = 0; i < 30; i++)
   temp[i] = i * 10.0;

for (i = 0; i < 20; i++) {
   for (j = 0; j < 3; j++) {
      mesh[i][j] = ++k;
   }
}

/* Create a vgroup with write access, then name it "Vertices". */
vgroup_id = Vattach(file_id, -1, "w");
status = Vsetname(vgroup_id, "Vertices");
```

```
/* Create a vdata to store the x,y values, set its name and class. */
vdata_id = VSattach(file_id, -1, "w");
status = VSsetname(vdata_id, "PX and PY");
status = VSsetclass(vdata_id, "Node List");

/* Specify the PX field information. */
status = VSfdefine(vdata_id, "PX", DFNT_FLOAT32, 2);

/* Specify the PY field information. */
status = VSfdefine(vdata_id, "PY", DFNT_FLOAT32, 2);

/* Set the field names. */
status = VSsetfields(vdata_id, "PX,PY");

/* Write the buffered data into the vdata object. */
num_of_elements = VSwrite(vdata_id, (VOIDP)pxy, 30, FULL_INTERLACE);

/* Insert the vdata into the vgroup. */
vg_position = Vinsert(vgroup_id, vdata_id);

/* Detach from the vdata. */
status = VSdetach(vdata_id);

/* Create a vdata to store the temperature property data. */
vdata_id = VSattach(file_id, -1, "w");
status = VSsetname(vdata_id, "PLIST");
status = VSsetclass(vdata_id, "Connectivity List");
status = VSfdefine(vdata_id, "PLIST", DFNT_FLOAT32, 1);
status = VSsetfields(vdata_id, "PLIST");
num_of_elements = VSwrite(vdata_id, (VOIDP)temp, 30, FULL_INTERLACE);
vg_position = Vinsert(vgroup_id, vdata_id);
status = VSdetach(vdata_id);

/* Create a vdata to store the mesh. */
vdata_id = VSattach(file_id, -1, "w");
status = VSsetname(vdata_id, "TMP");
status = VSsetclass(vdata_id, "Property List");
status = VSfdefine(vdata_id, "TMP", DFNT_INT8, 3);
status = VSsetfields(vdata_id, "TMP");
num_of_elements = VSwrite(vdata_id, (VOIDP)mesh, 20, FULL_INTERLACE);
vg_position = Vinsert(vgroup_id, vdata_id);
status = VSdetach(vdata_id);

/* Terminate access to the "Vertices" vgroup. */
status = Vdetach(vgroup_id);

/* Terminate access to the V interface. */
status = Vend(file_id);

/* Close the HDF file. */
status = Hclose(file_id);

}
```

FORTRAN:

```
      PROGRAM VDATA INSERT

       integer*4 file_id, vgroup_id, vdata_id, status, num_of_elements
       integer*4 vg_position
       integer*4 mesh(20, 3)
```

```
      double precision pxy(30, 2), temp(30)
      integer i, j
      integer hopen, vfatch, vfsnam, vsfatch
      integer vsfsfld, vsfwrit, vsfdtch, vsffdef, vfdtch
      integer hclose, vfinsrt, vfstart, vfend, vsfscls, vsfsnam
      data    pxy / -1.5, 2.3, -1.5, 1.98, -2.4, .67,
     +             -3.4, 1.46, -.65, 3.1, -.62, 1.23,
     +            -.4, 3.8, -3.55, 2.3, -1.43, 2.44,
     +            .23, 1.13, -1.4, 5.43, -1.4, 5.8,
     +            -3.4, 3.85, -.55, .3, -.21, 1.22,
     +            -1.44, 1.9, -1.4, 2.8, .94, 1.78,
     +            -.4, 2.32, -.87, 1.99, -.54, 4.11,
     +            -1.5, 1.35, -1.4, 2.21, -.22, 1.8,
     +            -1.1, 4.55, -.44, .54, -1.11, 3.93,
     +            -.76, 1.9, -2.34, 1.7, -2.2, 1.21 /

      integer*4 DFACC_CREAT
      parameter (DFACC_CREAT = 4)

C     Open an HDF file with full access.
C      DFACC_CREAT is defined in hdf.inc.
      file_id = hopen('Example2.hdf', DFACC_CREAT, 0)

C     Create a vgroup with write access, then name it 'Vertices'.
      status = vfstart(file_id)

C     Initialize the data buffer arrays.
      do 10 i = 1, 30
        temp(i) = i * 10.0
10    continue

      do 20 i = 1, 20
        do 30 j = 1, 3
          mesh(i, j) = k
             k = k + 1
30      continue
20    continue
      vgroup_id = vfatch(file_id, -1, 'w')
      status = vfsnam(vgroup_id, 'Vertices')

C     Create a vdata to store the x,y values.  The number
C     '5' in the third argument of vsffdef corresponds to
C     DFNT_FLOAT32 in hdf.inc.
      vdata_id = vsfatch(file_id, -1, 'w')
      status = vsfsnam(vdata_id, 'PX,PY')
      status =  vsfscls(vdata_id, 'Node List')
      status = vsffdef(vdata_id, 'PX', 5, 1)
      status = vsffdef(vdata_id, 'PY', 5, 1)
      status = vsfsfld(vdata_id, 'PX,PY')
      num_of_elements = vsfwrit(vdata_id, pxy, 30, FULL_INTERLACE)
      vg_position = vfinsrt(vgroup_id, vdata_id)
      status = vsfdtch(vdata_id)

C     Create a vdata to store the temperature data.  The number
C     '5' in the third argument corresponds to DFNT_FLOAT32
C     in hdf.inc.
      vdata_id = vsfatch(file_id, -1, 'w')
      status = vsfsnam(vdata_id, 'TMP')
      status =  vsfscls(vdata_id, 'Property List')
      status = vsffdef(vdata_id, 'TMP', 5, 1)
      status = vsfsfld(vdata_id, 'TMP')
      num_of_elements = vsfwrit(vdata_id, temp, 30, FULL_INTERLACE)
      vg_position = vfinsrt(vgroup_id, vdata_id)
```

```
      status = vsfdtch(vdata_id)

C     Create a vdata to store the mesh. The number '24' in the
C     third argument of vsffdef corresponds to DFNT_INT32 in
C     hdf.inc.
      vdata_id = vsfatch(file_id, -1, 'w')
      status = vsfsnam(vdata_id, 'PLIST')
      status = vsfscls(vdata_id, 'Connectivity List')
      status = vsffdef(vdata_id, 'PLIST', 24, 3)
      status = vsfsfld(vdata_id, 'PLIST')
      num_of_elements = vsfwrit(vdata_id, mesh, 20, FULL_INTERLACE)
      vg_position = vfinsrt(vgroup_id, vdata_id)
      status = vsfdtch(vdata_id)

C     Terminate access to the vgroup and the file.
      status = vfdtch(vgroup_id)
      status =  vfend(file_id)

C     Close the HDF file.
      status = hclose(file_id)

      end
```

EXAMPLE 3.

Adding a Scientific Data Set to a Vgroup

The following examples create an HDF file, an SDS and a vgroup. Then they insert the SDS into the vgroup.

C:

```
#include "hdf.h"
#include "mfhdf.h"
#include <stdio.h>

main()
{
   int ret;
   int32 fid, sds_id, sref;
   int32 dims[1];
   int32 vfid, vg_id, status;

   dims[0] = 10;

   vfid = Hopen("Example3.hdf", DFACC_CREATE, 0);
   fid = SDstart("Example3.hdf", DFACC_RDWR);
   sds_id = SDcreate(fid, "Test_SD1", DFNT_INT8, 1, dims);

   status = Vstart(vfid);
   vg_id = Vattach(vfid, -1, "w");

   sref = SDidtoref(sds_id);
   status = Vaddtagref(vg_id, DFTAG_NDG, sref);

   status = SDendaccess(sds_id);
   status = SDend(fid);

   status = Vdetach(vg_id);
   status = Vend(vfid);
```

```
    status = Hclose(vfid);

}
```

FORTRAN:

```
      PROGRAM ADD SDS TO VGROUP

       integer status
       integer*4 fid, sds_id, sref, vfid, vg_id
       integer dims(1)
       integer*4 DFACC_CREATE, DFACC_RDWR, DFNT_INT8, DFTAG_NDG
       parameter (DFACC_CREATE = 4, DFACC_RDWR = 3, DFNT_INT8 = 20,
      +           DFTAG_NDG = 720)

       integer hopen, sfstart, sfcreate, vfstart, vfatch
       integer sfid2ref, vfadtr, sfendacc, vfdtch, vfend, hclose

       dims(1) = 10
       vfid = hopen('Example3.hdf', DFACC_CREATE, 0)
       fid = sfstart('Example3.hdf', DFACC_RDWR)
       sds_id = sfcreate(fid, 'Test_SD1', DFNT_INT8, 1, dims)
       status = vfstart(vfid)
       vg_id = vfatch(vfid, -1, 'w')

       sref = sfid2ref(sds_id)
       status = vfadtr(vg_id, DFTAG_NDG, sref)

       status = sfendacc(sds_id)

       status = vfdtch(vg_id)
       status = vfend(vfid)
       status = hclose(vfid)

       end
```

1.6 Reading from Vgroups

Reading from vgroups is more complicated than writing to vgroups. The process of reading from vgroups involves three steps:

1. Locate the appropriate vgroup.
2. Identify the members of the vgroup of interest.
3. Perform the read operation on the vgroup members.

1.6.1 Locating Vgroups in Files

There are several routines provided for the purpose of searching for a particular vgroup. The syntax for these routines are as follows.

C:

```
num_of_lones = Vlone(file_id, ref_array, maxsize);
ref_num = Vgetid(file_id, vgroup_ref);
status = Vgetname(vgroup_id, vgroup_name);
status = Vgetclass(vgroup_id, vgroup_class);
```

FORTRAN:

```
num_of_lones = vflone(file_id, ref_array, maxsize)
ref_num = vfgetid(file_id, vgroup_ref)
status = vfgfnam(vgroup_id, vgroup_name)
status = vfgcls(vgroup_id, vgroup_class)
```

1.6.1.1 Locating a Lone Vgroup: Vlone

A ***lone vgroup*** is one that is not a member of another vgroup. **Vlone** returns an array of reference numbers for all lone vgroups in a file. **Vlone** is useful for locating unattached vgroups in a file or the vgroups at the top of a grouping hierarchy. The parameter `ref_array` is an array allocated to hold the reference numbers. The `maxsize` argument specifies the maximum size of `ref_array`. At most `maxsize` reference numbers will be returned in `ref_array`. The value returned from **Vlone** is the total number of vgroups that are not linked to any other vgroup in the file.

The space allocated for `ref_array` depends on how many lone vgroups are expected to be found. A size of 32,767 integers is adequate to handle any case. To use dynamic memory instead of allocating such a large array, first call **Vlone** with `maxsize` set to a small value, for example, `0` or `1`, then use the returned value to allocate memory for `ref_array`. This array is then to be passed to an argument of **Vlone**.

1.6.1.2 Determining the Next Vgroup Identifier: Vgetid

Vgetid sequentially searches through an HDF file to check vgroup reference numbers. It returns the reference number for the vgroup immediately following the vgroup with the reference number in the `vgroup_ref` parameter. To initiate a search, **Vgetid** may be called with `vgroup_ref` set to `-1`. Doing so returns the reference number of the first vgroup in the file. Any attempt to search past the last vgroup in a file will cause **Vgetid** to return a value of `-1`.

1.6.1.3 Determining a Vgroup's Name and Class: Vgetname and Vgetclass

Vgetname returns the name of the vgroup through the parameter `vgroup_name`. The maximum length of the name is defined by the macro `VGNAMELENMAX`.

Vgetclass returns the class of the vgroup through the parameter `vgroup_class`. The maximum length of the class name is defined by the macro `VGNAMELENMAX`.

TABLE 5D **Vlone, Vgetid, Vgetname and Vgetclass Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
Vlone (**vflone**)	file_id	int32	integer	File identifier.
	ref_array	int32	integer	Buffer for the reference numbers of lone vgroups.
	maxsize	int32	integer	Maximum number of vgroups to store in `ref_array`.
Vgetid (**vfgid**)	file_id	int32	integer	File identifier.
	vgroup_ref	int32	integer	Reference number of the previous vgroup.
Vgetname (**vfgnam**)	vgroup_id	int32	integer	Vgroup identifier.
	vgroup_name	char *	character* (*)	Buffer for the name of the vgroup.
Vgetclass (**vfgcls**)	vgroup_id	int32	integer	Vgroup identifier.
	vgroup_class	char *	character* (*)	Buffer for the vgroup class. (if any)

EXAMPLE 4. **Printing Vgroup Reference Numbers**

These examples obtain and print the reference numbers for all of the lone vgroups and the vdata objects in an HDF file.

C:

```
#include "hdf.h"
```

```
main( )
{

int32 file_id, status;
int32 vgroup_ref, vgroup_id;
int32 maxsize, i, num_of_lones;
int32 n_entries, *ref_array;
char vgroup_name[VGNAMELENMAX];

/* Open the "Example2.hdf" file. */
file_id = Hopen("Example2.hdf", DFACC_READ, 0);

/* Initialize HDF for subsequent vgroup/vdata access. */
status = Vstart(file_id);

/* Get and print the reference numbers of all the lone
vgroups.  First, call Vlone with maxsize set to 0 to
get the length of the storage array, then call Vlone
again to put the reference id numbers into the array. */
maxsize = Vlone(file_id, ref_array, 0);
ref_array = (int32 *) HDmalloc(sizeof(int32) * maxsize);
num_of_lones = Vlone(file_id, ref_array, maxsize);

for (i = 0; i < maxsize; i++)
   printf("Lone vgroup reference id  %d\n", ref_array[i]);

printf("*******\n");
HDfree(ref_array);

/* Set the reference number variable to start the search
at the first vgroup in the file. */
vgroup_ref = -1;

/* Print every reference id in the file. */
while (TRUE) {
   vgroup_ref = Vgetid(file_id, vgroup_ref);
   if (vgroup_ref == -1) break;
   vgroup_id = Vattach(file_id, vgroup_ref, "r");
   status = Vinquire(vgroup_id, &n_entries, vgroup_name);
       printf("Found vgroup with ref %d, number of entries %d, name
%s\n",
              vgroup_ref, n_entries, vgroup_name);
       Vdetach(vgroup_id);
}

/* Terminate access to the vgroup interface and the file. */
status = Vend(file_id);
status = Hclose(file_id);

}
```

FORTRAN:

```
     PROGRAM PRINT REFS

      integer*4 file_id, ref_array(31)
      integer vgroup_ref, vgroup_id, status
      integer n_entries, i
      character vgroup_name(64)
      logical loop_flag
      integer hopen, vflone, vfgid, vfatch, vfdtch
      integer vfinq, hclose, vfstart, vfend
```

```
      integer*4 DFACC_READ
      parameter (DFACC_READ = 1)

C     Open an HDF file with read access.
C     DFACC_READ is defined in hdf.inc.
      file_id = hopen('Example2.hdf', DFACC_READ, 0)

C     Initialize HDF for subsequent vgroup/vdata access.
      status = vfstart(file_id)

C     Get and print the reference numbers of all the lone
C     vgroups.
      status = vflone(file_id, ref_array, 30)
      do 10 i = 1, status
         print *, 'Lone vgroup reference id ', ref_array(i)
10    continue

      print *, '*******'

C     Set the reference number variable to start the search
C     at the first vgroup in the file.
      vgroup_ref = -1

C     Print every reference id in the file.
      loop_flag = .TRUE.
      if (loop_flag) then
20       vgroup_ref = vfgid(file_id, vgroup_ref)
        if (vgroup_ref .eq. -1) then
          go to 30
        end if
        vgroup_id = vfatch(file_id, vgroup_ref, 'r')
        status = vfinq(vgroup_id, n_entries, vgroup_name)
        print *, 'Found vgroup with ref ', vgroup_ref,
     +           ' number of entries ', n_entries,
     +           ' name ', vgroup_name
        status = vfdtch(vgroup_id)
        goto 20
      end if

C     Terminate access to the V interface and the file.
30    status = vfend(file_id)
      status = hclose(file_id)

      end
```

1.7 Vgroup Attributes

Version 4.1r1 of HDF includes the capability of assigning attributes to a vgroup. The concept of attributes is fully explained in Chapter 3, titled *Scientific Data Sets (SD API)*. To review briefly: an attribute has a name, a data type, a number of attribute values and the attribute values themselves. All attribute values must be of the same data type - for example, an integer cannot be added to an attribute value consisting of ten characters, or a character value cannot be included in an attribute value consisting of two 32-bit integers.

Any number of attributes can be assigned to a vgroup - however, each attribute name must be unique among all attributes in the vgroup.

1.7.1 Querying Information on a Given Vgroup Attribute: Vattrinfo

Vattrinfo returns the name, data type, number of values, and the size of the values of the specified attribute of the specified vgroup.

The values of the *attr_name*, *data_type*, *count* and *size* parameters can be set to `NULL`, if the information returned by these parameters are not needed.

The value of *attr_index* parameter is used as the index of the target vgroup attribute, and is zero-based. For example, a *attr_index* value of `4` would refer to the fifth attribute of the vgroup.

Vattrinfo returns `SUCCEED` if successful, and `FAIL` otherwise.

TABLE 5E **Vattrinfo Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
Vattrinfo **(vfainfo)**	vgroup_id	int32	integer	Vgroup identifier.
	attr_index	intn	integer	Index of the target attribute..
	name	char *	character* (*)	Returned name of the target attribute.
	data_type	int32 *	integer	Returned data type of the target attribute.
	count	int32 *	integer	Returned number of values of the target attribute.
	size	int32 *	integer	Returned size, in bytes, of the values of the target attribute.

1.7.2 Querying the Vgroup Version of a Given Vgroup: Vgetversion

Vgetversion returns the vgroup version number of the specified vgroup. The structure of the vgroup has gone through several changes since HDF was first written. Determining the version of any particular vgroup is necessary as some of the older versions of vgroups don't support attributes.

The newest version is version 4, which is the only version that supports vgroup attributes. **Vgetversion** returns a value of `VSET_NEW_VERSION` when encountering a vgroup of this vgroup version. Version 3 is the vgroup version corresponding to all versions of the HDF library between 3.2 and 4.0 release 2. **Vgetversion** returns a value of `VSET_VERSION` when encountering a vgroup of this vgroup version. The third version is version 2. This vgroup version corresponds to all HDF library versions before version 3.2, and **Vgetversion** returns a value of `VSET_OLD_VERSION` when encountering a vgroup of this vgroup version.

Vgetversion returns the vset version number if successful, and `FAIL` otherwise.

TABLE 5F **Vgetversion Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
Vgetversion **(vfgver)**	vgroup_id	int32	integer	Vgroup identifier.

1.7.3 Querying the Total Number of Vgroup Attributes: Vnattrs

Vnattrs returns number of attributes assigned to this vgroup and its fields when successful, and `FAIL` otherwise.

TABLE 5G **Vnattrs Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
Vnattrs (vfnatts)	vgroup_id	int32	integer	Vgroup identifier.

1.7.4 Querying the Values of a Given Vgroup Attribute: Vgetattr

Vgetattr returns all of the values of the specified attribute of the specified vgroup.

The *attr_index* parameter is the ordinal, zero-based index of the target attribute.

Vgetattr returns `SUCCEED` if successful, `FAIL` otherwise.

TABLE 5H **Vgetattr Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
Vgetattr (vsgnatt/vsgcatt)	vgroup_id	int32	integer	Vgroup identifier.
	attr_index	intn	integer	Index of the target attribute.
	values	VOIDP	<valid numeric data type>	Buffer containing attribute values.

1.7.5 Setting the Attribute of a Vgroup: Vsetattr

Vsetattr attaches an attribute to a vgroup. If the attribute already exists, the new values will replace the current ones, provided the data type and order have not been changed. If either the data type or the order have been changed, **Vsetattr** will exit on an error condition.

Vsetattr returns `SUCCEED` if successful, `FAIL` otherwise.

TABLE 5I **Vsetattr Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
Vsetattr (vfsatt/vfscatt)	vgroup_id	int32	integer	Vgroup identifier.
	attr_name	char *	character* (*)	Name of the attribute.
	data_type	int32	integer	Data type of the attribute.
	count	int32	integer	Number of values the attribute contains.
	values	VOIDP	<valid numeric data type>	Buffer containing the attribute values.

1.7.6 Retrieving the Index of a Vgroup Attribute Given the Attribute Name:

Vfindattr

Vfindattr returns the index of an attribute with the given name, specified by the value of the *attr_name* parameter, of a vgroup.

Vfindattr returns the index of the target attribute if successful, and `FAIL` otherwise.

TABLE 5J

Vfindattr Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
Vfindattr (vffdatt)	vgroup_id	int32	integer	Vgroup identifier.
	attr_name	char *	character* (*)	Name of the target attribute.

EXAMPLE 5.

Attaching and Querying Vgroup Attributes

These examples attach an attribute to the vgroup created in Example 1 (in the file named “Example1.hdf”). Then it checks the number of attributes in the file, the version of the vgroup and then reads the attribute data.

C:

```
#include "hdf.h"

#define FILE_NAME "Example1.hdf"
#define VGATTR_NAME "Vgroup Attribute 1"

main( )
{

int32 file_id, vgroup_ref, vgroup_id, status;
int32 vg_version;
int32 v_type, v_count, v_size;
char vg_attr[6] = {'m','N','p', 'S', 't', '\0'};
char vgattr_buf[6], vattrname[30];

/* Open the HDF file. */
file_id = Hopen(FILE_NAME, DFACC_RDWR, 0);

/* Initialize the V interface. */
status = Vstart(file_id);

/* Get the reference number of the target vgroup. */
vgroup_ref = Vfind(file_id, "VG_Name_1");

/* Attach to the target vgroup. */
vgroup_id = Vattach(file_id, vgroup_ref, "w");

/* Get the version of the vgroup just created. */
vg_version = Vgetversion(vgroup_id);

/* Attach an attribute to the vgroup. */
status = Vsetattr(vgroup_id, VGATTR_NAME, DFNT_CHAR,
            6, vg_attr);

/* Get information about the vgroup attribute. */
status = Vattrinfo(vgroup_id, 0, vattrname, &v_type,
            &v_count, &v_size);
```

```
/* Get the vgroup attribute. */
status = Vgetattr(vgroup_id, 0, vgattr_buf);

/* Detach from the vgroup, close the V interface and the file. */
status = Vdetach(vgroup_id);
status = Vend(file_id);
status = Hclose(file_id);

}
```

FORTRAN:

```
         PROGRAM CREATE QUERY ATTRS

      integer file_id, vgroup_ref, vgroup_id, status
      integer vg_version
      integer hopen, vfstart, vfind, vfatch, vfdtch, hclose
      integer vfend, vsgver, vfgcatt, vfainfo, vfscatt
      integer v_type, v_count, v_size
      character*20 vgattrname, vgattr_buf
      character* (*) filename, vgattr_name, vg_attr
      parameter (filename = 'Example1.hdf',
     +   vgattr_name = 'Vgroup Attribute 1',
     +   vg_attr = 'mnpst'
     +    )

C     The following parameters are defined in hdf.inc.
      integer DFACC_RDWR, DFNT_CHAR
      parameter (DFACC_RDWR = 3, DFNT_CHAR = 4)

C     Open an HDF file with full access.
      file_id = hopen(filename, DFACC_RDWR, 0)

C     Initialize the V interface.
      status = vfstart(file_id)

C     Get the reference number of the target vgroup.
      vgroup_ref = vfind(file_id, 'VG_Name_1')

C     Attach to the target vdata.
      vgroup_id = vfatch(file_id, vgroup_ref, 'w')

C     Get the version of the vgroup just created.
      vg_version = vsgver(vgroup_id)

C     Attach an attribute to the vgroup.
      status = vfscatt(vgroup_id, vgattr_name, DFNT_CHAR, 5, vg_attr)

C     Get information about the vgroup attribute.
      status = vfainfo(vgroup_id, 0, vgattrname, v_type, v_count,
     +                 v_size)

C     Get the vgroup attribute.
      status = vfgcatt(vgroup_id, 0, vgattr_buf)

C     Detach from the vgroup, close the V interface and the file.
      status = vfdtch(vgroup_id)
      status = vfend(file_id)
      status = hclose(file_id)

      end
```

1.8 Obtaining Information About the Contents of a Vgroup

When the target vgroup has been located, a way to determine its internal structure is needed. The HDF routines necessary to perform this operation are listed here, along with their syntax:

C:

```
num_of_elems = Vntagrefs(vgroup_id);
status = Vgettagref(vgroup_id, index, tag, ref);
num_of_pairs = Vgettagrefs(vgroup_id, tag_array, ref_array, max-
                                size);
true_false = Vinqtagref(vgroup_id, tag, ref);
true_false = Visvg(vgroup_id, vgroup_ref);
true_false = Visvs(vgroup_id, vdata_ref);
```

FORTRAN:

```
num_of_elems = vfntr(vgroup_id)
status = vfgttr(vgroup_id, index, tag, ref)
num_of_pairs = vfgttrs(vgroup_id, tag_array, ref_array, maxsize)
true_false = vfinqtr(vgroup_id, tag, ref)
true_false = vfisvg(vgroup_id, vgroup_ref)
true_false = vfisvs(vgroup_id, vdata_ref)
```

Each routine queries a specific item of vgroup information. The parameters of these routines are described below. (See Table 5K on page 164.)

1.8.1 Returning the Tag/Reference Number Pairs of Vgroup Members: Vntagrefs, Vgettagref and Vgettagrefs

Vntagrefs returns the number of members stored in the specified vgroup. This routine is used together with **Vgettagrefs** or in a loop with **Vgettagref** to identify the HDF objects linked to a given vgroup.

Vgettagref returns the tag/reference number pair of a specified HDF object stored within the vgroup in the `tag` and `ref` parameters respectively. The parameter `index` specifies the location of the member within the vgroup and is zero-based.

Vgettagrefs returns the tags/reference number pairs of the vgroup members in arrays. The tag/reference number pairs are returned in the parameters `tag_array` and `ref_array`. The parameter `maxsize` specifies the maximum number of tag/reference number pairs to return, therefore each array must be at least `maxsize` in size.

1.8.2 Returning Vgroup Member Information: Vinqtagref, Visvg and Visvs

Vinqtagref returns `TRUE` (or `1`) if the data object with tag/reference number pair specified by the `tag` and `ref` parameters are a member of the vgroup and `FALSE` (or `0`) otherwise.

Visvg returns `TRUE` (or `1`) if the vgroup reference number specified by the `vgroup_ref` parameter is that of a vgroup stored in the vgroup, and `FALSE` (or `0`) otherwise.

Visvs returns `TRUE` (or `1`) if the vdata reference number specified by the `vdata_ref` parameter is that of a vdata stored in the vgroup, and `FALSE`(or `0`) otherwise.

TABLE 5K

Vgetname, Vgetclass, Visvg and Visvs Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
Vntagrefs (vfntr)	vgroup_id	int32	integer	Vgroup identifier.
Vgettagref (vfgttr)	vgroup_id	int32	integer	Vgroup identifier.
	index	int32	integer	Index of the tag/reference number pair tobe retrieved.
	tag	int32	integer	Pointer to the tag.
	ref	int32	integer	Pointer to the reference number.
Vgettagrefs (vfgttrs)	vgroup_id	int32	integer	Vgroup identifier.
	tag_array	int32 *	integer	Buffer for the returned tags.
	ref_array	int32 *	integer	Buffer for the returned reference numbers.
	maxsize	int32	integer	Maximum number of tag/reference numbrr pairs to be returned.
Vinqtagref (vfinqtr)	vgroup_id	int32	integer	Vgroup identifier.
	tag	int32	interger	Tag of item to be queried.
	ref	int32	integer	Reference number of items to be queried.
Visvg (vfisvg)	vgroup_id	int32	integer	Vgroup identifier.
	vgroup_ref	int32	integer	Vgroup reference number to be queried.
Visvs (vfisvs)	vgroup_id	int32	integer	Vgroup identifier.
	vdata_ref	int32	integer	Vdata reference number to be queried.

EXAMPLE 6.

Extracting the Tag/Reference Number Pairs of Vgroup Members

The following examples make use of **Vntagrefs** to determine the number of tag/reference number pairs assigned to the members of a vgroup. It then uses **Vgettagref** to extract each tag/reference number pair.

C:

```
#include "hdf.h"

main( )
{
int32 file_id;
int32 vgroup_id, vgroup_ref;
int32 vdata_tag, vdata_ref;
int32 status, i, npairs;

/* Open the "Example2.hdf" file. */
file_id = Hopen("Example2.hdf", DFACC_READ, 0);

/* Initialize HDF for subsequent vgroup/vdata access. */
status = Vstart(file_id);

/* Attach to every vgroup in the file. */
vgroup_ref = -1;

while (TRUE) {
   vgroup_ref = Vgetid(file_id, vgroup_ref);
   if (vgroup_ref == -1) break;
   vgroup_id = Vattach(file_id, vgroup_ref, "r");

   /* Get the total number of tag/reference id pairs. */
   npairs = Vntagrefs(vgroup_id);
```

```
   /* Print every tag and reference id with their
   corresponding file position. */
   for (i = 0; i < npairs; i++) {
      status = Vgettagref(vgroup_id, i, &vdata_tag, &vdata_ref);
      printf("Found tag = %d, ref = %d at position %d.\n", \
              vdata_tag, vdata_ref, i+1);
   }

   /* Terminate access to the vgroup. */
   status = Vdetach(vgroup_id);
}

/* Terminate access to the V interface and close the file. */
status = Vend(file_id);
status = Hclose(file_id);

}
```

FORTRAN:

```
     PROGRAM EXTRACT TAG

      integer*4 file_id, vgroup_id, vgroup_ref, vdata_tag
      integer*4 vdata_ref, npairs
      integer i, status
      integer hopen, vfatch, vfntr, vfgttr
      integer vfgid, vfdtch, hclose, vfstart, vfend
      integer*4 DFACC_READ
      parameter (DFACC_READ = 1)

C     Open an HDF file with read-only access.
C     DFACC_READ is defined in hdf.inc.
      file_id = hopen('Example2.hdf', DFACC_READ, 0)

C     Initialize HDF for subsequent vgroup/vdata access.
      status = vfstart(file_id)

C     Attach to every vgroup in the file.
      vgroup_ref = -1
30    vgroup_ref = vfgid(file_id, vgroup_ref)

      if (vgroup_ref .eq. -1) then
        go to 20
      else
        vgroup_id = vfatch(file_id, vgroup_ref, 'r')
C       Get the total number of tag/reference id pairs.
        npairs = vfntr(vgroup_id)
C       Print every tag and reference id with their
C       corresponding file position.
        do 10 i = 1, npairs
          status = vfgttr(vgroup_id, i-1, vdata_tag, vdata_ref)
          if (status .eq. -1) then
            go to 40
          end if
          print *, 'Found tag = ', vdata_tag, ' ref = ', vdata_ref,
     +             ' at position ', i

10        continue

C       Terminate access to the vgroup.
40      status = vfdtch(vgroup_id)
      end if
```

```
      go to 30

C     Terminate access to the V interface and close the file.
20    status = vfend(file_id)
      status = hclose(file_id)

      end
```

1.9 The Vgroup Command-Line Utilities

The **vshow** and **vmake** utilities are Unix command line utilities for working with vgroups and vdatas. We give a very brief description of these here. For more information, refer to Chapter 13, titled *HDF Command-Line Utilities*.

1.9.1 Vshow

The **vshow** utility lists information about vgroups and vdatas in HDF files. It lists and displays information on all the vgroups in the file in the order stored, followed by the vdatas.

1.9.2 Vmake

The **vmake** utility is a general utility for generating vsets in an HDF file. **Vmake** is used to create new vgroups, store data into new vdatas and create links among vgroups and vdatas.

1.10 Obsolete Vgroup Interface Routines

The following routines have been replaced by newer routines with similar functionality. They are still supported by the HDF V interface, but their use is not recommended. HDF may not support these routines in a future version.

1.10.1 Determining the Next Vgroup or Vdata Identifier: Vgetnext

Vgetnext searches through a vgroup for vgroups or vdatas. The syntax for **Vgetnext** is:

C: `ref_num = Vgetnext(vgroup_id, vgroup_ref);`

FORTRAN: `ref_num = vfgnxt(vgroup_id, vgroup_ref)`

Vgetnext searches the vgroup with the identifier `vgroup_id` and returns the reference number of the vgroup or vdata following the vgroup with the reference number specified by the `vgroup_ref` parameter. To initiate the search, **Vgetnext** is called with `vgroup_ref` set to `-1`. This will return the reference number of the first object in the target vgroup. The value `-1` is returned when an error occurs or when there are no more entities in the host vgroup.

Vgetnext is now obsolete as the routine **Vgettagref** provides the same functionality and, in addition, is not restricted to searching for members that are vgroups or vdatas.

1.10.2 Determining the Number of Members and Vgroup Name: Vinquire

The syntax for **Vinquire** is:

C:
```
status = Vinquire(vgroup_id, n_members, vgroup_name);
```

FORTRAN:
```
status = vfinq(vgroup_id, n_members, vgroup_name)
```

Vinquire returns the number of data objects and the name of the vgroup specified by `vgroup_id` in the `n_members` and `vgroup_name` parameters respectively. If either `n_members` or `vgroup_name` are set to `NULL`, the corresponding data is not returned. The maximum length of the vgroup's name is defined by the macro `VGNAMELENMAX`.

Vinquire is now obsolete as the **Vntagrefs** routine can be used to get the number of objects in a vgroup and **Vgetname** can be used to retrieve the name of a vgroup.

TABLE 5L **Vgetnext and Vinquire Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
Vgetnext (vfgnxt)	vgroup_id	int32	integer	Vgroup identifier of the parent vgroup.
	vgroup_ref	int32	integer	Reference number for the target vgroup.
Vinquire (vfinq)	vgroup_id	int32	integer	Vgroup identifier.
	members	int32 *	integer	Pointer to the number of entries in the vgroup.
	vgroup_name	char *	character* (*)	Buffer for the name of the vgroup.

1.11 Vgroup Backward Compatibility Issues

1.11.1 Vset Implementation Integrated into the Vgroup Interface

In HDF versions before 2.0, vsets were created and manipulated through an interface separate from the other main HDF interfaces like the SDS interface, VS interface, etc.. A pointer to a specially-defined vset structure was returned by the vset interface and data type definitions specific to the vset interface was used. The names of these definitions were prefaced by "LOCAL_".

After HDF version 2.0 the `vs_id` identifier replaced the pointer to the vset structure, the "LOCAL_" data type definitions were made obsolete by newer definitions and all vset functionality was integrated in the V interface routines. HDF programs written for HDF libraries earlier than version 2.0 should be modified accordingly in order to function correctly with newer versions of HDF.

Chapter 6

8-Bit Raster Images (DFR8 API)

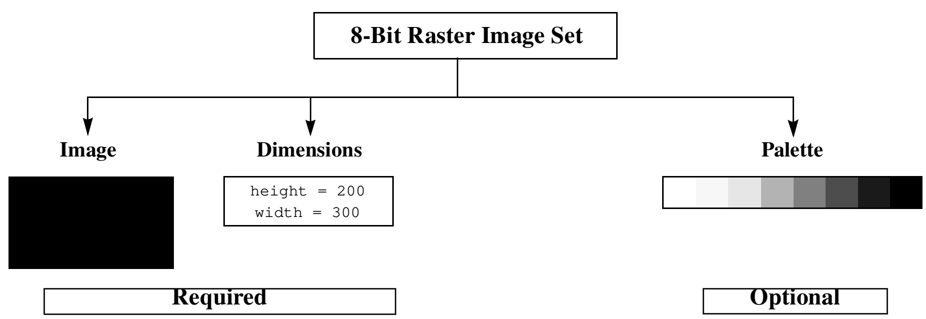
6.1 Chapter Overview

This chapter describes the 8-bit raster image data model and the single-file DFR8 interface routines. The DFR8 interface is a single-file interface that consists of routines for reading and writing raster image sets.

6.2 The 8-Bit Raster Data Model

The data model for the ***8-bit raster image*** set, or ***RIS8***, an acronym for "Raster Image Set, 8-bit", supports three types of objects; two-dimensional 8-bit raster images, dimensions and palettes. The latter two items occur once per RIS8. The following figure shows the contents of an 8-bit raster image set.

FIGURE 6a **8-Bit Raster Image Set Contents**



6.2.1 Required 8-Bit Raster Image Data Set Objects

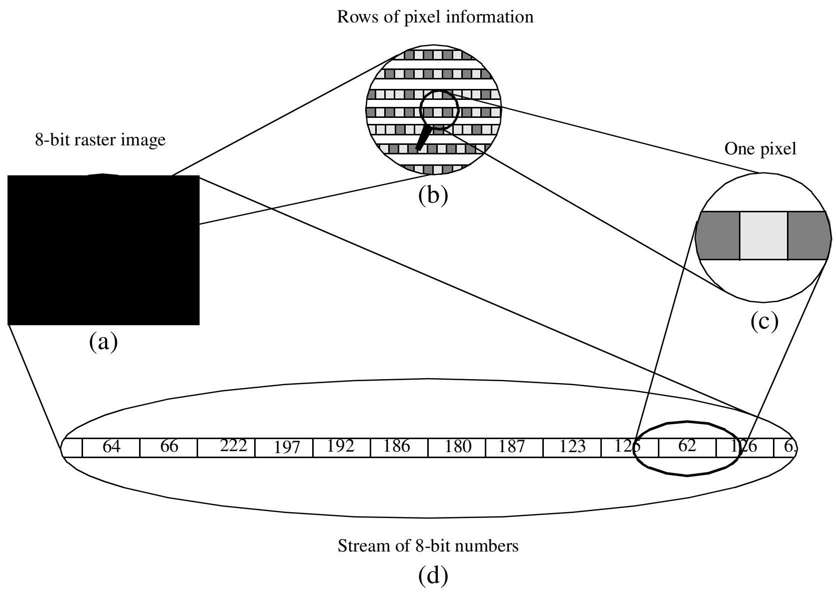
Every RIS8 object requires an image and dimension object. Required objects are created by the HDF library using information provided at the time the image is written.

6.2.1.1 8-Bit Raster Image Data Representation

An ***8-bit raster image*** is a two-dimensional array of 8-bit numbers which represent ***pixels*** or "picture elements".The first row of pixels corresponds to the top row of the image, the second row of pixels to the second row of the image and so forth. Pixel values range from 0 to 255, and indicate to the hardware which colors to use when mapping the corresponding pixels to the screen display. A ***color lookup table***, or ***palette***, provides the means of correlating pixel values to colors.

As an example, consider a stream of 8-bit numbers representing a raster image. (See Figure 6b.) When the image is displayed, the color associated with the first number in the data stream is placed in the upper left corner of the image. The remainder of the first line is then painted from left-to-right using as many values from the data stream as is necessary to complete the line. The remainder of the rows are similarly painted from left-to-right and top-to-bottom until every value in the data stream appears is represented by one pixel in the image.

FIGURE 6b **The Data Representation of an 8-Bit Raster Image**



An 8-bit raster image (a) is a set of rows displayed from left-to-right consisting of rows of pixel information (b) where each pixel (c) is represented by values stored as a single stream of 8-bit numbers (d).

6.2.1.2 8-Bit Raster Image Dimension

The ***dimensions*** of an image are its height and width in pixels.

6.2.2 Optional 8-Bit Raster Image Data Set Objects

6.2.2.1 Palettes

A ***palette*** is a lookup table consisting of 256 unique numerical values, each of which map to the 256 possible pixel color values and is stored in a RIS8 object. For more details on HDF palettes refer to Chapter 9, titled *Palettes (DFP API)*.

6.2.3 Compression Method

The compression method indicates if and how the image is compressed. It can be, at the programmer's option, explicitly set or left as its default setting of no compression. Compression schemes supported by HDF version 4.0 are ***run-length encoding*** or ***RLE***, ***joint photographic expert group compression***, or ***JPEG***, and ***image compression***, or ***IMCOMP*** . The list of compression methods is presented below. (See Table 6A.) The HDF tags `COMP_RLE`, `COMP_IMCOMP` and `COMP_JPEG` are respectively defined as the values `11`, `12` and `2` in the "hcomp.h" header file.

TABLE 6A

8-Bit Raster Image Compression Method List

Compression Method	Type	Compression Code	Requirements
None	N/A	COMP_NONE	Image data only (default setting).
RLE	Lossless	COMP_RLE	Image data only.
JPEG	Lossy	COMP_JPEG	Image data, quality factor and compatibility factor.
IMCOMP	Lossy	COMP_IMCOMP	Image data and palette.

RLE Compression

The ***RLE*** method is a lossless compression method recommended for images where data retention is critical. The RLE algorithm compresses images by condensing strings of identical pixel values into two bytes. The first byte identifies the number of pixels in the string and the second byte records the pixel value for the string.

The amount of space saved by RLE depends upon how much repetition there is among adjacent pixels. If there is a great deal of repetition, more space is saved and if there is little repetition, the savings can be very small. In the worst case when every pixel is different from the one that precedes it an extra byte is added for every 127 bytes in the image.

JPEG Compression

The ***JPEG***, or Joint Photographic Expert Group, compression method is a lossy compression algorithm whose use is recommended for photographic or scanned images. Using JPEG compression to reduce the size of an image changes the values of the pixels and hence may alter the meaning of the corresponding data. Version 5.0 of the JPEG library is available in HDF version 4.0.

JPEG compression requires two parameters, the first the level of image quality and the second, compatibility. The ***quality factor*** determines how much of the data will be lost and thus directly impacts the size of the compressed image. A quality factor of 1 specifies the lowest quality or maximum image compression. A quality factor of 100 specifies the highest quality or minimum image compression. Note that all images compressed using the JPEG algorithm are stored in a lossy manner, even those stored with a quality factor of 100. Usually, it is best to experiment with the quality factor to find the most acceptable one.

The `baseline` parameter determines whether the contents of the quantization tables used during compression are forced into the range of 0 to 255. The `baseline` parameter is normally set to the value `1` which forces baseline results. You should set the value of the `baseline` parameter to values other than `1` *only* if you are familiar with the JPEG algorithm.

IMCOMP Compression

IMCOMP is a lossy compression method available in earlier versions of HDF. IMCOMP compression is generally of inferior quality to JPEG compression and is not recommended unless your images will be viewed on a 16-color monitor. For backward compatibility, IMCOMP compression is supported in the HDF library, For details on IMCOMP refer to Appendix F, titled *Backward Compatibility Issues*.

6.3 The 8-Bit Raster Image Interface

The HDF library contains routines for reading and writing 8-bit raster image sets. The functions **DFR8addimage**, **DFR8putimage** and **DFR8getimage** are sufficient for most reading and writing operations.

6.3.1 8-Bit Raster Image Library Routines

The names of all C functions in the 8-bit raster image interface are prefaced by "DFR8" and the names of the equivalent Fortran-77 functions are prefaced by "d8". These routines are divided into the following categories:

- ***Write routines*** create raster image sets and store them in new files or append them to existing files.
- ***Read routines*** determine the dimensions and palette assignment for an image set, read the actual image data and provide sequential or random read access to any raster image set.

The DFR8 function calls are further defined in Table 6B and in the *HDF Reference Guide*.

TABLE 6B **DFR8 Library Routines**

Category	Routine Name		Description
	C	Fortran-77	
Write	DFR8writeref	d8wref	Stores the raster image using the specified reference number.
	DFR8setpalette	d8spal	Sets palette for multiple 8-bit raster images.
	DFR8addimage	d8aimg	Appends an 8-bit raster image to a file.
	DFR8putimage	d8pimg	Writes an 8-bit raster image to an existing file or creates the file.
	DFR8setcompress	d8setcomp	Sets the compression type.
	None	d8sjpeg	Passes the quality and compatibility factors needed for the JPEG compression algorithm.
Read	DFR8getdims	d8gdims	Retrieves dimensions for an 8-bit raster image.
	DFR8getimage	d8gimg	Retrieves an 8-bit raster image and its palette.
	DFR8readref	d8rref	Gets the next raster image with the specified reference number.
	DFR8restart	d8first	Ignores information about last file accessed and restarts from beginning.
	DFR8nimages	d8nims	Returns number of raster images in a file.
	DFR8lastref	d8lref	Returns reference number of the last element accessed.
	DFR8getpalref	None	Returns the reference number of the palette associated with the last image accessed.

6.4 Writing 8-Bit Raster Images

The DFR8 programming model for writing an 8-bit raster image sets is as follows:

1. Set the compression type if the image is to be compressed. (optional)

2. Identify the palette if one is to be stored with the image. (optional)
3. Write the raster data to the file.

The two optional steps can be invoked in any order, as long as they are executed before Step 3. By default, images are stored uncompressed with no associated palette.

6.4.1 Storing a Raster Image: DFR8putimage and DFR8addimage

To write a raster image to an HDF file, the calling program must contain the following:

C: `status = DFR8putimage(filename, image, width, height, compress);`

FORTRAN: `status = d8pimg(filename, image, width, height, compress)`

OR

C: `status = DFR8addimage(filename, image, width, height, compress);`

FORTRAN: `status = d8aimg(filename, image, width, height, compress)`

DFR8putimage and **DFR8addimage** write an 8-bit raster image to an HDF file named by the `filename` parameter. When given a new filename, **DFR8putimage** and **DFR8addimage** create a new file and write the raster image as the first raster image in the file. When given an existing filename, **DFR8putimage** overwrites the file whereas **DFR8addimage** appends data to the end of the file.

In the **DFR8putimage** and **DFR8addimage** functions, the raster data is passed in the `image` parameter and the width and height of the image are passed in the `width` and `height` parameters. The compression algorithm used to store the image is passed in the compress parameter. Valid compress values include `COMP_NONE`, `COMP_RLE`, `COMP_JPEG` and `COMP_IMCOMP`. `COMP_NONE` represents no compression (storage only), `COMP_RLE` represents run-length encoding, `COMP_JPEG` represents JPEG compression and `COMP_IMCOMP` represents IMCOMP encoding.

Parameters for **DFR8putimage** and **DFR8addimage** are further described below. (See Table 6C on page 173.)

TABLE 6C **DFR8putimage and DFR8addimage Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFR8putimage (d8pimg) and DFR8addimage (d8aimg)	filename	char *	character* (*)	Name of file the raster image will be stored in.
	image	VOIDP	<valid numeric data type>	Image data array.
	width	int32	integer	Number of columns in the raster image.
	height	int32	integer	Number of rows in the raster image.
	compress	int16	integer	Compression type.

EXAMPLE 1. **Writing an 8-Bit Raster Image to an HDF File**

In the following code examples, **DFR8addimage** and **d8aimg** are used to write an 8-bit image to a file named "Example1.hdf". Note that the order in which the dimensions for the image array are declared differs between C and Fortran-77.

C:

```
#include "hdf.h"
```

```
#define WIDTH 5
#define HEIGHT 6

main( )
{

/* Initialize the image array */
static uint8 raster_data[HEIGHT][WIDTH] =
    { 1, 2, 3, 4, 5,
    6, 7,8, 9, 10,
    11, 12, 13,14, 15,
    16, 17, 18, 19, 20,
    21, 22, 23,24, 25,
    26, 27, 28,29, 30 };

intn status;


/* Write the 8-bit raster image to file */
status = DFR8addimage("Example1.hdf", raster_data,
                WIDTH, HEIGHT, 0);

}
```

FORTRAN:

```
      PROGRAM RASTER8

      character*1 raster_data(5,6)
      integer retn, d8aimg

      integer*4 WIDTH, HEIGHT
      parameter(WIDTH = 5, HEIGHT = 6)

C     Initialize the image array
      data raster_data /  1,  2,  3,  4,  5,
     $                    6,  7,  8,  9, 10,
     $                   11, 12, 13, 14, 15,
     $                   16, 17, 18, 19, 20,
     $                   21, 22, 23, 24, 25,
     $                   26, 27, 28, 29, 30 /

C     Write the 8-bit raster image to the file
      retn = d8aimg('Example1.hdf', raster_data, WIDTH, HEIGHT, 0)

      end
```

6.4.2 Adding a Palette to an RIS8 Object: DFR8setpalette

DFR8setpalette identifies the palette to be used for the subsequent write operations. It may be used to assign a palette to a single image or several images. After a palette has been set, it acts as the current palette until it is replaced by another call to **DFR8setpalette**. To create a raster image set containing a palette, the calling program must contain the following:

C:

```
status = DFR8setpalette(palette);
status = DFR8addimage(filename, image, width, height, compress);
```

FORTRAN:

```
status = d8spal(palette)
status = d8aimg(filename, image, width, height, compress)
```

DFR8setpalette takes `palette` as its only parameter. To set the default palette to “no palette”, pass `NULL` as the `palette` parameter. **DFR8setpalette** is further defined in the following table.

TABLE 6D

DFR8setpalette Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFR8setpalette (d8spal)	palette	uint8 *	character* (*)	Palette to be assigned.

EXAMPLE 2.

Writing a Palette and an Image in RIS8 Format

These examples demonstrate how a palette stored in the array `colors` and the raw image stored in the 20 x 20 array `picture` is written to a RIS8 object. The image is not compressed and, in these examples, uninitialized. The raster image set is stored as the first image in "Example2.hdf". Note that because **DFR8putimage** recreates the file, anything previously contained in this file will be erased.

C:

```
#include "hdf.h"

#define WIDTH 20
#define HEIGHT 20

main( )
{
uint8 colors[256*3], picture[HEIGHT][WIDTH];
uint8 i, j;
int16 status;

/* Initialize image arrays. */
for (j = 0; j < WIDTH; j++) {
    for (i = 0; i < HEIGHT; i++)
        picture[j][i] = 1;
      }

/* Set the current palette. */
status = DFR8setpalette(colors);

/* Write the image data to the file. */
status = DFR8putimage("Example2.hdf", picture, WIDTH,
                HEIGHT, COMP_NONE);

}
```

FORTRAN:

```
     PROGRAM WRITE UNCOMPRESSED RIS8

      integer   d8spal, d8pimg, status, i, j
      integer   colors(768)
      integer*4   WIDTH, HEIGHT, COMP_NONE
      parameter (COMP_NONE = 0,
     +           WIDTH = 20,
     +           HEIGHT = 20)
      integer   picture(WIDTH, HEIGHT)

C     Initialize the image data.
      do 20 j = 1, WIDTH
```

```
          do 10 i = 1, HEIGHT
             picture(j, i) = 1
10        continue
20      continue

C       Set the current palette.
        status = d8spal(colors)

C       Write the image data to the file.
        status = d8pimg('Example2.hdf', picture, WIDTH, HEIGHT,
     +                  COMP_NONE)

        end
```

6.4.3 Compressing 8-Bit Raster Image Data: DFR8setcompress

The compression type is determined by the tag passed as the fifth argument in calls to the **DFR8putimage** and **DFR8addimage** routines. **DFR8setcompress** is currently required only to reset the default JPEG compression options. However, future versions of this routine will support additional compression schemes.

To set non-default compression parameters, the calling program should contain the following sequence of routines:

C:

```
status = DFR8setcompress(type, c_info);
status = DFR8addimage(filename, image, width, height, compress);
```

FORTRAN:

```
status = d8scomp(type)
<compression-specific code>
status = d8aimg(filename, image, width, height, compress)
```

Notice that the calling sequence for C differs from the calling sequence for Fortran-77. Once the compression is set, the parameter `type` in the **DFR8setcompress** routine, or **d8scomp** in Fortran-77, specifies the compression method that will be used when storing the raster images. However, the `c_info` parameter, which is a pointer to a structure that contains information specific to the compression scheme indicated by the `type` parameter in **DFR8setcompress**, is missing from **d8scomp**. Because data structures of variable size are not supported in Fortran-77, another routine specific to the compression library is required in the Fortran-77 calling sequence.

The `c_info` union is described in Chapter 3, titled *Scientific Data Sets (SD API)*. The values contained in this union are passed into the **d8sjpeg** Fortran-77-specific routine.

Parameters for **DFR8setcompress** and **d8sjpeg** are further described in Table 6E below.

TABLE 6E **DFR8setcompress Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFR8setcompress (d8scomp)	type	int32	integer	Compression method.
	c_info	comp_info *	None	Pointer to JPEG information structure.
d8sjpeg	quality	none	integer	JPEG quality factor.
	baseline	none	integer	JPEG baseline.

EXAMPLE 3.

Writing a Set of Compressed 8-Bit Raster Images

These examples contain a series of calls in which four 20 x 20 images are written to the same file. The first two use palette `paletteA` and are compressed using the RLE method; the third and fourth use palette `paletteB` and are not compressed.

C:

```
#include "hdf.h"

#define WIDTH 20
#define HEIGHT 20

main ( )
{
uint8 paletteA[256*3], paletteB[256*3];
uint8 picture1[HEIGHT][WIDTH], picture2[HEIGHT][WIDTH];
uint8 picture3[HEIGHT][WIDTH], picture4[HEIGHT][WIDTH];
uint8 i, j;
int16 status;

/* Initialize image arrays. */
for (j = 0; j < WIDTH; j++) {
    for (i = 0; i < HEIGHT; i++) {
        picture1[j][i] = 1;
        picture2[j][i] = 1;
        picture3[j][i] = 1;
        picture4[j][i] = 1;
    }
}

/* Set the first palette. */
status = DFR8setpalette(paletteA);

/* Write the compressed image data to the HDF file. */
status = DFR8putimage("Example3.hdf", (VOIDP)picture1, WIDTH, HEIGHT, \
                COMP_RLE);
status = DFR8addimage("Example3.hdf", (VOIDP)picture2, WIDTH, HEIGHT, \
                COMP_RLE);

/* Set the second palette. */
status = DFR8setpalette(paletteB);

/* Write the uncompressed image data to the HDF file. */
status = DFR8addimage("Example3.hdf", (VOIDP)picture3, WIDTH, HEIGHT, \
                COMP_NONE);
status = DFR8addimage("Example3.hdf", (VOIDP)picture4, WIDTH, HEIGHT, \
                COMP_NONE);

}
```

FORTRAN:

```
      PROGRAM WRITE IMAGE SETS

        integer d8spal, d8pimg, d8aimg, status
        integer*4 COMP_RLE, COMP_NONE, WIDTH, HEIGHT
        parameter (COMP_RLE = 11,
     +           COMP_NONE = 0,
     +           WIDTH = 20,
     +           HEIGHT = 20)
```

```
      integer paletteA(768), paletteB(768)
      integer picture1(WIDTH, HEIGHT), picture2(WIDTH, HEIGHT)
      integer picture3(WIDTH, HEIGHT), picture4(WIDTH, HEIGHT)

C     Initialize the image data.
      do 20 j = 1, WIDTH
       do 10 i = 1, HEIGHT
        picture1(j, i) = 1
        picture2(j, i) = 1
        picture3(j, i) = 1
        picture4(j, i) = 1
10     continue
20    continue

C     Set the first palette.
      status = d8spal(paletteA)

C     Write the compressed image data to the HDF file.
      status = d8pimg('Example3.hdf', picture1, WIDTH, HEIGHT,
     +               COMP_RLE)
      status = d8aimg('Example3.hdf', picture2, WIDTH, HEIGHT,
     +               COMP_RLE)

C     Set the second palette.
      status = d8spal(paletteB)

C     Write the uncompressed image data to the HDF file.
      status = d8aimg('Example3.hdf', picture3, WIDTH, HEIGHT,
     +               COMP_NONE)
      status = d8aimg('Example3.hdf', picture4, WIDTH, HEIGHT,
     +               COMP_NONE)

      end
```

EXAMPLE 4.

Compressing and Writing a 8-Bit Raster Image

In the following examples, **DFR8addimage** and **DFR8compress** are used to compress an 8-bit image and write it to an HDF file named "Example2.hdf". Notice that compressing an image in C requires only one function call, whereas compressing an image using Fortran-77 requires two. The second Fortran-77 call is required because it is not valid to pass a structure as a parameter in Fortran-77.

C:

```
#include "hdf.h"
#include "hcomp.h"

#define WIDTH 3
#define HEIGHT 5
#define PIXEL_DEPTH 3

main( )
{
/* Initialize the image array. */
static uint8 raster_data[HEIGHT][WIDTH][PIXEL_DEPTH] =
    {  1, 2, 3,  4, 5, 6,  7, 8, 9,
      10,11,12, 13,14,15, 16,17,18,
      19,20,21, 22,23,24, 25,26,27,
      28,29,30, 31,32,33, 34,35,36,
      37,38,39, 40,41,42, 43,44,45 };
```

```
static comp_info compress_info;
intn status;

/* Initialize JPEG compression structure. */
compress_info.jpeg.quality = 60;
compress_info.jpeg.force_baseline = 1;

/* Set JPEG compression for storing the image. */
status = DFR8setcompress(COMP_JPEG, &compress_info);

/* Write the 8-bit image data to file. */
status = DFR8addimage("Example2.hdf", (VOIDP)raster_data, WIDTH,
                          HEIGHT, COMP_JPEG);

}
```

FORTRAN:

```
     PROGRAM COMPRESS RIS8

      integer d8aimg, d8scomp, d8sjpeg, status
      integer*4 WIDTH, HEIGHT, PIXEL_DEPTH, COMP_JPEG

C     COMP_JPEG is defined in hcomp.h.
      parameter(WIDTH = 3,
     +          HEIGHT = 5,
     +          COMP_JPEG = 1,
     +          PIXEL_DEPTH = 3)
      character raster_data(PIXEL_DEPTH, WIDTH, HEIGHT)

C     Initialize the image array.
      data raster_data
     + /  1, 2, 3,  4, 5, 6,  7, 8, 9,
     +    10,11,12, 13,14,15, 16,17,18,
     +    19,20,21, 22,23,24, 25,26,27,
     +    28,29,30, 31,32,33, 34,35,36,
     +    37,38,39, 40,41,42, 43,44,45  /

C     Set compression.
      status = d8scomp(COMP_JPEG)

C     Set JPEG parameters to quality = 60, and turn compatibility on.
      status = d8sjpeg(60, 1)

C     Write the 8-bit image data to the HDF file.
      status = d8aimg('Example2.hdf', raster_data, WIDTH, HEIGHT,
     +              COMP_JPEG)

      end
```

6.4.4 Specifying the Reference Number of an RIS8: DFR8writeref

DFR8writeref specifies the reference number of the image to be written when **DFR8addimage** or **DFR8putimage** is called. Use the following calling sequence to invoke **DFR8writeref**:

C:

```
status = DFR8writeref(filename, ref);
status = DFR8addimage(filename, image, width, height, compress);
```

FORTRAN:

```
status = d8wref(filename, ref)
status = d8aimg(filename, image, width, height, compress)
```

DFR8writeref assigns the reference number passed in the `ref` parameter to the next image the file specified by the `filename` parameter. If the value of `ref` is the same as the reference number of an existing RIS8, the existing raster image data will be overwritten. The parameters for **DFR8writeref** are further described below. (See Table 6F.)

It is unlikely that you will need this routine, but if you do, use it with caution. It is not safe to assume that a reference number indicates the file position of the corresponding image as there is no guarantee that reference numbers appear in sequence in an HDF file.

TABLE 6F **DFR8writeref Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFR8writeref (**d8wref**)	filename	char *	character* (*)	Name of the HDF file containing the raster image.
	ref	uint16	integer	Reference number for next call to **DFR8getimage**.

6.5 Reading 8-Bit Raster Images

The DFR8 programming model for reading an 8-bit raster image set is as follows:

1. Determine the dimensions of the image if they are not known prior to the read operation.
2. Read the image from the file.

6.5.1 Reading a Raster Image: DFR8getimage

If dimensions of the image are known, **DFR8getimage** is the only function call needed to read a raster image. If a file is being opened for the first time, **DFR8getimage** returns the first image in the file. Additional calls will return successive images in the file, therefore images are read in the order which they were written to the file. **DFR8getdims** is called before **DFR8getimage** so that space allocations for the image and palette can be checked and the dimensions verified. If this information is already known, **DFR8getdims** may be omitted.

To read a raster image from an HDF file, the calling program must contain the following:

C: `status = DFR8getimage(filename, image, width, height, palette);`

FORTRAN: `status = d8gimg(filename, image, width, height, palette)`

DFR8getimage retrieves the next 8-bit image from the HDF file name specified by the `filename` parameter. If the image in the file is compressed, **DFR8getimage** first decompresses it then places it in memory at the location pointed to by the `image` parameter. The dimensions of the array allocated to hold the image are specified by the `width` and `height` parameters and may be larger than the actual image.The palette, if present, is stored in memory at the location pointed to by the `palette` parameter. If it contains a `NULL` value the palette is not loaded, even if there is one stored with the image. The parameters for **DFR8getimage** are defined further in Table 6G below.

Notice that in Example 4, as in the case of **DFR8addimage**, the order in which the dimensions for the `image` array are declared differs between C and Fortran-77. Fortran-77 declarations require the width before the height while the C declaration requires the height before the width as Fortran-77 arrays are stored in column-major order, while C arrays are stored in row-major order. (row-major order implies that the second coordinate varies fastest). When **d8gimg** reads an image from a file, it assumes column-major order.

6.5.2 Querying the Dimensions of an 8-Bit Raster Image: DFR8getdims

DFR8getdims opens a named file, finds the next image or the first image if the file is being opened for the first time, retrieves the dimensions of the image and determines if there is a palette associated 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. In this way, images are read in the same order in which they were written to the file.

To determine the dimensions of an image before attempting to read it, the calling program must include the following routines:

C:
```
status = DFR8getdims(filename, width, height, haspalette);
status = DFR8getimage(filename, image, width, height, palette);
```

FORTRAN:
```
status = d8gdim(filename, width, height, haspalette)
status = d8gimg(filename, image, width, height, palette)
```

DFR8getdims retrieves dimension and palette information about the next 8-bit image in the file specified by `filename`. The returned information is pointed to by the `width` and `height` parameters. The `haspalette` parameter determines the presence of a palette and returns a value of `1` if it exists and `0` otherwise. The parameters for **DFR8getdims** are defined further in the following table.

TABLE 6G **DFR8getdims and DFR8getimage Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFR8getdims (d8gdims)	filename	char *	character* (*)	Name of the HDF file containing the set of raster images.
	width	int32 *	integer	Number of columns in the next raster image.
	height	int32 *	integer	Number of rows in the next raster image.
	ispalette	intn *	integer	“1” if a palette exists, otherwise “0”.
DFR8getimage (d8gimg)	filename	char *	character* (*)	Name of HDF file with the raster image.
	image	uint8 *	character* (*)	Buffer for the raster image.
	width	int32	integer	Width of the raster image buffer.
	height	int32	integer	Height of the raster image buffer.
	palette	uint8 *	character* (*)	Palette assigned to the raster image.

EXAMPLE 5. **Reading an 8-Bit Raster Image**

The following examples search the "Example1.hdf" file created in Example1 for the dimensions of an 8-bit image. Although the **DFR8getdims** call is optional, it is included as a demonstration of how to check the dimensions of an image. This example also assumes that the data set does not include a palette, therefore `NULL` is passed as the palette parameter. If the palette argument is `NULL` (or "0" in Fortran-77), all palette data is ignored.

C:
```
#include "hdf.h"
```

```
#define WIDTH 5
#define HEIGHT 6

main( )
{
uint8 raster_data[HEIGHT][WIDTH];
int32 width, height;
intn haspal, status;

/* Get the dimensions of the image */
status = DFR8getdims("Example1.hdf", &width, &height, &haspal);

/* Read the raster data if the dimensions are correct */
if (width <= WIDTH && height <= HEIGHT)
    status = DFR8getimage("Example1.hdf", (VOIDP)raster_data, width,
                height, NULL);

}
```

FORTRAN:

```
     PROGRAM RASTER8

      character*1 image(5, 6)
      integer status, height, width, d8gimg, d8gdims, haspal
      integer*4 width, height

C     Get the dimensions of the image.
      status = d8gdims('Example1.hdf', width, height, haspal)

C     Read the raster data if the dimensions are correct.
      if (width .le. 5 .and. height .le. 6) then
         status = d8gimg('Example1.hdf', image, width, height, 0)
      endif

      end
```

6.5.3 Reading an Image with a Given Reference Number: DFR8readref

DFR8readref accesses specific images that are stored in files containing multiple raster image sets. It is an optionally used before **DFR8getimage** to set the access pointer to the specified raster image. **DFR8readref** can be used in connection with vgroups, which identify their members by tag/reference number pairs. See Chapter 5, titled *Vgroups (V API)*, for a discussion of vgroups and tag/reference number pairs.

To access a specific raster image set, use the following calling sequence:

C:
```
status = DFR8readref(filename, ref);
status = DFR8getimage(filename, image, width, height, palette);
```

FORTRAN:
```
status = d8rref(filename, ref)
status = d8gimg(filename, image, width, height, palette)
```

DFR8readref specifies that the target for the next read operation performed on the HDF file specified by the `filename` parameter is the object with the reference number named in the `ref` parameter. The parameters required for **DFR8readref** are defined further in the following table.

TABLE 6H

DFR8readref Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFR8readref (d8rref)	filename	char *	character* (*)	Name of HDF file containing the raster image.
	ref	uint16	integer	Reference number for next call to **DFR8getimage**.

6.5.4 Specifying the Next 8-Bit Raster Image to be Read: DFR8restart

DFR8restart causes the next call to **DFR8getimage** or **DFR8getdims** to read the first raster image set in the file. Use the following call to invoke **DFR8restart**:

C:
```
status = DFR8restart( );
```

FORTRAN:
```
status = d8first( )
```

6.6 8-Bit Raster Image Information Retrieval Routines

6.6.1 Querying the Total Number of 8-Bit Raster Images: DFR8nimages

DFR8nimages returns the total number of 8-bit raster image sets in a file and has the following syntax:

C:
```
num_of_images = DFR8nimages(filename);
```

FORTRAN:
```
num_of_images = d8nimg(filename)
```

TABLE 6I

DFR8nimages Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFR8nimages (d8nims)	filename	char *	character* (*)	Name of the HDF file.

6.6.2 Determining the Reference Number of the Most-Recently-Accessed 8-Bit Raster Image: DFR8lastref

DFR8lastref returns the reference number most recently used in writing or reading an 8-bit raster image. This routine is primarily used for attaching annotations to images and adding images to vgroups. (See Chapters 8, titled *Annotations (DFAN API)* and Chapter 5, titled *Vgroups (V API)* for more detailed information on how to use reference numbers in connection with these applications.)

The following calling sequence uses **DFR8lastref** to find the reference number of the 8-bit raster image most recently added to an HDF file:

C:
```
status = DFR8addimage(filename, image, width, height, compress);
lastref = DFR8lastref( );
```

FORTRAN:
```
status = d8aimg(filename, image, width, height, compress)
lastref = d8lref( )
```

DFR8putimage or **DFR8getimage** can be used instead of **DFR8addimage** with similar results.

6.6.3 Determining the Reference Number of the Palette of the Most-Recently-Accessed 8-Bit Raster Image: DFR8getpalref

DFR8getpalref returns the reference number of the palette associated with the most recently used in writing or reading an 8-bit raster image. The **DFR8getdims** routine must be called before **DFR8getpalref**, as **DFR8getdims** initializes internal structures required by **DFR8getpalref**.

There is currently no Fortran-77 version of the **DFR8getpalref** routine.

TABLE 6J **DFR8nimages Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFR8getpalref	pal_ref	uint16 *	None	Pointer to the returned reference number of the palette.

6.7 RIS8 Backward Compatability Issues

6.7.1 Attribute "long_name" Included in HDF for netCDF Compatibility

In several routines of the RIS8 interface, the value returned by `label` is the value of the attribute named "long_name" and that the value returned by `coordsys` is the value of the attribute named "cordsys".

This was done in order to provide HDF with the ability to read netCDF files. While this aspect of HDF functionality will not affect its ability to read HDF data files written by programs compiled with earlier versions of HDF, it is advisable for HDF users to know this to be aware of the significance of the "long_name" and "cordsys" attribute names in a list of attributes.

6.7.2 Raster Image Group Implementation with New RIS8 Tags

As HDF has evolved, a variety of structures have been used to store raster image sets. For instance, HDF first began grouping 8-bit raster images together with dimensions and palettes by insuring that their reference numbers fell in a certain pattern. This method of organizing raster images quickly lead to very complicated collections of images, dimension records, and palettes, and eventually was replaced by an specific grouping structure known as a ***Raster Image Group***, or ***RIG***, with a completely new set of tags.

To maintain backward compatibility with older versions of HDF, the RIS8 interface supported by HDF version 4.1 and later recognizes raster images stored using either set of HDF tags. Details on the different tags and structures used to store raster images can be found in the *HDF Specification Manual*, versions 3.2 or later.

Chapter 7

24-Bit Raster Images (DF24 API)

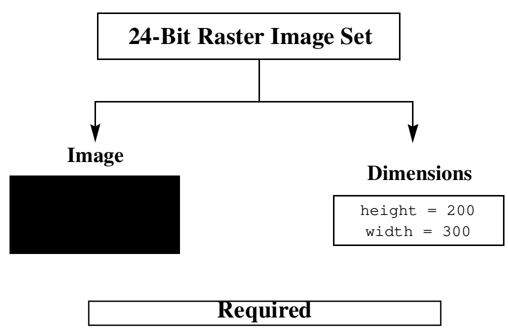
7.1 Chapter Overview

This chapter describes the 24-bit raster data model and the single-file DF24 routines available for storing and retrieving 24-bit raster images.

7.2 The 24-Bit Raster Data Model

The ***24-bit raster image set***, or ***RIS24***, data model supports two primary data objects: two-dimensional 24-bit raster images and dimensions. The primary member of the set is the ***24-bit raster image***, a two-dimensional array of pixels or picture elements. Each pixel is represented by three 8-bit numbers of image data. An optional compression method describes the method used, if any, to compress the image. Figure 7a shows the contents of a 24-bit raster image set.

FIGURE 7a **24-Bit Raster Image Set Contents**



7.2.1 Required 24-Bit Raster Image Data Set Objects

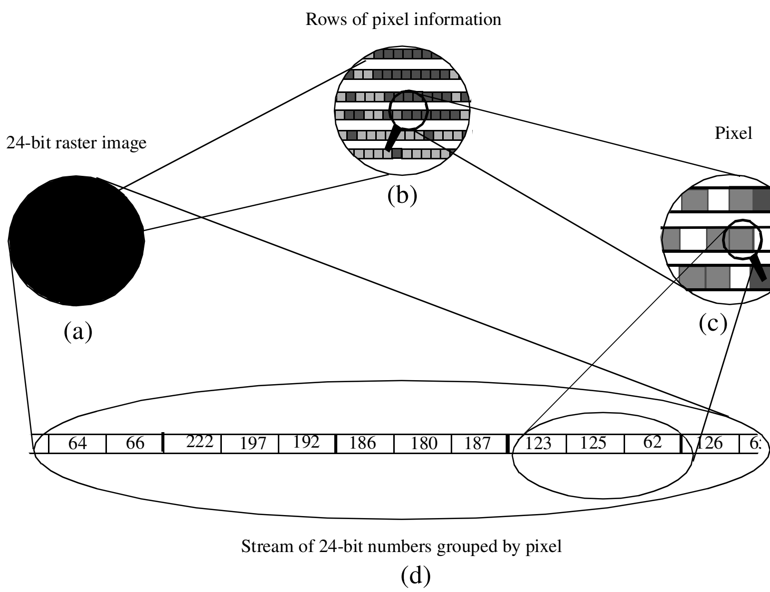
All 24-bit raster images must contain image data and a dimension record. These objects are created by the HDF library using information provided at the time the image is written to file.

7.2.1.1 24-Bit Raster Image Data Representation

The 24-bit raster image object contains a set of 24-bit pixel values, each of which has three 8-bit components; one for the red, one for the green, and one for the blue color component of the image. These values, referred to as ***RGB values***, are arranged in one of three specific ways, as described in Section 7.2.2.2 on page 187. The pixel values are arranged in rows, painted from left-to-right, top-to-bottom. As each pixel in a 24-bit image is represented in the image data by three 8-bit numbers, palettes are unnecessary and are not included in the 24-bit raster data model.

As an example, consider a stream of 24-bit numbers representing a raster image (Fig. 4.1a). To display the image, the color associated with the first number in the data stream appears in the upper left corner of the image. The remainder of the first line is then painted from left-to-right using as many values from the data stream as necessary to complete the line. The remainder of the rows are similarly painted from left-to-right, top-to-bottom until every value in the data stream appears as one pixel in the image.

FIGURE 7b **The Numerical Representation of a 24-Bit Raster Image**



A 24-bit raster image (a) is a set of rows displayed from left-to-right consisting of rows of pixels (b) whose values are stored as three 8-bit numbers (c) in a stream of data (d). In this figure, the image is interleaved by pixel.

7.2.1.2 24-Bit Raster Image Dimension

The ***dimensions*** of an image are the height and width of the image in pixels.

7.2.2 Optional 24-Bit Raster Image Data Set Objects

7.2.2.1 Compression Method

The only 24-bit compression method currently available in HDF is the JPEG algorithm . The applicable HDF compression tags are `COMP_JPEG`, and `COMP_NONE.` (See Table 7A.) The HDF tags `COMP_JPEG and COMP_NONE are` defined as the values `2`, and `0` respectively in the "hdf.h" header file.

TABLE 7A

24-Bit Raster Image Compression Method List

Compression Method	Type	Compression Code	Requirements
None	Lossless	COMP_NONE	Image data only (default setting).
JPEG	Lossy	COMP_JPEG	Image data, quality factor and compatibility factor.

JPEG Compression

The ***JPEG*** compression method is a lossy compression algorithm whose use is recommended for photographic or scanned images. Using JPEG compression to reduce the size of an image changes the values of the pixels and therefore may alter the meaning of the corresponding data.

For more information on the JPEG algorithm, refer to Chapter 6, titled *8-bit Raster Images (DFR8 API)*.

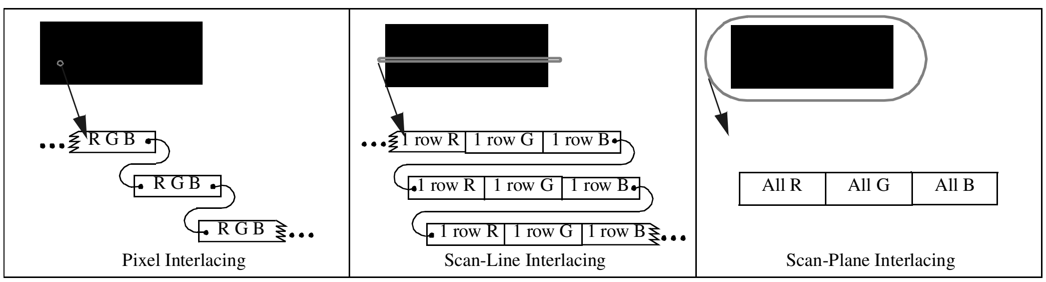
7.2.2.2 Interlace Modes

Because graphics applications and hardware devices vary in the way they access image data, HDF supports three interlace formats. By storing an image using a format that is consistent with the expected application or device, it is possible to achieve substantial improvements in performance.

HDF provides three options for organizing the color components in 24-bit raster images. These options consist of pixel interlacing, scan-line interlacing, and scan-plane interlacing. (See Figure 7c.) Storing the color components grouped by pixel, as in red-green-blue, red-green-blue, etc., is called ***pixel interlacing***. Storing the color components by line, as in one row of red, one row of green, one row of blue, one row red, etc., is called ***scan-line interlacing***. Finally, storing the color components grouped by color plane, as in the red components first, the green components second, and the blue components last, is called ***scan-plane interlacing***. Unless otherwise specified, the HDF 24-bit raster model assumes that all 24-bit images are stored using pixel interlacing.

FIGURE 7c

RGB Interlace Format for 24-Bit Raster Images



An interlace format describes both the physical format of an image as it is stored in memory and in the file. When writing to a file, HDF stores a 24-bit image using the same interlace format as it has in memory. However, when reading from a file, it is possible to make the in-core interlacing mode different from that used in the file. The following table contains a summary of the interlacing format available in the DF24 interface.

TABLE 7B

24-Bit Raster Image Interlace Format

format	Description	DF24setil or d2setil Parameter	Size of Image Array
Pixel	Components grouped by pixel.	DFIL_PIXEL	Width x Height x 3

Scan-line	Components grouped by row.	DFIL_LINE	Width x 3 x Height
Scan-plane	Components grouped by plane.	DFIL_PLANE	3 x Width x Height

7.3 The 24-Bit Raster Interface

The HDF library currently contains several routines for storing 24-bit raster images in the HDF format. The **DF24addimage**, **DF24putimage**, and **DF24getimage** routines are sufficient for most reading and writing operations.

7.3.1 24-Bit Raster Image Library Routines

The names of all C routines in the 24-bit raster image interface are prefaced by "DF24". The equivalent Fortran-77 routines are prefaced by "d2". These routines are divided into the following categories:

- ***Write routines*** create raster image sets and store them in new files or append them to existing files.
- ***Read routines*** determine the dimensions and interlace format of an image set, read the actual image data, and provide sequential or random read access to any raster image set.

The DF24 function calls are more explicitly defined in Table 7C and on their respective pages in the *HDF Reference Guide*.

TABLE 7C

DF24 Library Routines

Purpose	Routine Name		Description
	C	Fortran-77	
Write	DF24addimage	d2aimg	Appends a 24-bit raster image to a file.
	DF24putimage	d2pimg	Writes a 24-bit raster image to file by overwriting all existing data.
	DF24setcompress	d2scomp	Sets the compression method for the next raster image written to the file.
	DF24setdims	d2sdims	Sets the dimensions for the next raster image written to the file.
	DF24setil	d2setil	Sets the interlace format of the next raster image written to the file.
	DF24lastref	d2lref	Reports the last reference number assigned to a 24-bit raster image.
	None	d2sjpeg	Fortran-specific routine for setting the parameters needed for the JPEG compression algorithm.
Read	DF24getdims	d2gdims	Retrieves the dimensions before reading the next raster image.
	DF24getimage	d2gimg	Reads the next 24-bit raster image.
	DF24reqil	d2reqil	Retrieves the interlace format before reading the next raster image.
	DF24readref	d2rref	Reads 24-bit raster image with the specified reference number.
	DF24restart	d2first	Returns to the first 24-bit raster image in the file.
	DF24nimage	d2nimg	Reports the number of 24-bit raster images in a file.

7.4 Writing 24-Bit Raster Images

The DF24 programming model for writing a 24-bit raster image set is as follows:

1. Set the interlace format if the interlacing is to be different from pixel interlacing. (optional)
2. Set the compression type if the image is to be compressed. (optional)
3. Write the raster data to the file.

Steps 1 and 2 can be invoked in any order, as long as they are executed before Step 3. By default, images are stored uncompressed using pixel interlacing.

7.4.1 Writing a 24-Bit Raster Image: DF24putimage and DF24addimage

To write a raster image to an HDF file, the calling program must contain one of the following function calls:

C: `status = DF24putimage(filename, image, width, height);`

FORTRAN: `status = d2pimg(filename, image, width, height)`

OR

C: `status = DF24addimage(filename, image, width, height);`

FORTRAN: `status = d2aimg(filename, image, width, height)`

DF24putimage and **DF24addimage** write a 24-bit raster images to the HDF file specified by the `filename` parameter. When given a new file name, **DF24putimage** and **DF24addimage** create a new file and write the raster image as the first raster image in the file. If a file with the specified filename exists, **DF24putimage** overwrites the previous contents of the file whereas **DF24addimage** appends data to the end of the file.

DF24putimage and **DF24addimage** passes the raster data in the `image` parameter and the width and height of the image in the `width` and `height` parameters. The array `image` is assumed to be the width times the height times three bytes in length for each color component. The parameters for **DF24putimage** and **DF24addimage** are further defined below. (See Table 7D on page 189.)

TABLE 7D **DF24putimage and DF24addimage Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DF24putimage (d2pimg)	filename	char *	character* (*)	Name of file to store the raster image.
	image	VOIDP	<valid numeric data type>	Raster image to be written.
	width	int32	integer	Number of columns in the image.
	height	int32	integer	Number of rows in the image.
DF24addimage (d2aimg)	filename	char *	character* (*)	Name of file to store the raster image.
	image	VOIDP	<valid numeric data type>	Raster image to be written.
	width	int32	integer	Number of columns in the image.
	height	int32	integer	Number of rows in the image.

EXAMPLE 1. **Writing a 24-Bit Raster Image to an HDF File**

In the following examples, **DF24addimage** and **d2aimg** are used to write a 24-bit image to an HDF file named "Example1.hdf." **DF24addimage** assumes row-major order. Therefore, the Fortran-77 declaration requires the width (rows) before the height (columns), whereas the C declaration requires the height before the width. The interlace format setting is the default (pixel interlacing).

C:

```
#include "hdf.h"

#define WIDTH 5
#define HEIGHT 6
#define PIXEL_DEPTH 3

main( )
{

/* Initialize the image array. */
static uint8 raster_data[HEIGHT][WIDTH][PIXEL_DEPTH] =
{  1, 2, 3,  4, 5, 6,  7, 8, 9,  10,11,12,  13,14,15,
  16,17,18, 19,20,21, 22,23,24,  25,26,27,  28,29,30,
  31,32,33, 34,35,36, 37,38,39,  40,41,42,  43,44,45,
  46,47,48, 49,50,51, 52,53,54,  55,56,57,  58,59,60,
  61,62,63, 64,65,66, 67,68,69,  70,71,72,  73,74,75,
  76,77,78, 79,80,81, 82,83,84,  85,86,87,  88,89,90 };
    intn status;

/* Write the 24-bit raster image to the HDF file. */
status = DF24addimage("Example1.hdf", (VOIDP)raster_data, WIDTH, \
                  HEIGHT);

}
```

FORTRAN:

```
     PROGRAM WRITE RIS24

      integer status, d2aimg
      integer*4 WIDTH, HEIGHT, PIXEL_DEPTH
      parameter (WIDTH = 5,
     +         HEIGHT = 6,
     +         PIXEL_DEPTH = 3)

      character raster_data(PIXEL_DEPTH, WIDTH, HEIGHT)

C     Initialize the image array.
      data raster_data
     +     /  1, 2, 3,  4, 5, 6,  7, 8, 9,  10,11,12,  13,14,15,
     +     16,17,18, 19,20,21, 22,23,24,  25,26,27,  28,29,30,
     +     31,32,33, 34,35,36, 37,38,39,  40,41,42,  43,44,45,
     +     46,47,48, 49,50,51, 52,53,54,  55,56,57,  58,59,60,
     +     61,62,63, 64,65,66, 67,68,69,  70,71,72,  73,74,75,
     +     76,77,78, 79,80,81, 82,83,84,  85,86,87,  88,89,90 /

C     Write the 24-bit raster image to the file.
      status = d2aimg('Example1.hdf', raster_data, WIDTH,
     +                HEIGHT)

      end
```

7.4.2 Setting the Interlace Format: DF24setil

DF24setil indicates the interlace format to be used for all subsequent write operations. **DF24setil** changes the default setting from pixel interlacing to the selected format. When the format is set, it acts as the default until it is reset by another call to **DF24setil**. To change the default interlace format , the calling program must contain the following routines:

C:
```
status = DF24setil(il);
status = DF24addimage(filename, image, width, height);
```

FORTRAN:
```
status = d2setil(il)
status = d2aimg(filename, image, width, height)
```

DF24setil takes `il` as its only parameter. Valid values for `il` are `DFIL_PIXEL`, `DFIL_LINE`, and `DFIL_PLANE`. The parameters for **DF24setil** are further defined below. (See Table 7E on page 192.)

EXAMPLE 2.

Writing 24-Bit Raster Images Using Scan-plane Interlacing

In the following examples, **DF24addimage** is used to write a 24-bit image to an HDF file called "Example2.hdf". The **DF24setil** function used here to change the default format setting from pixel interlacing to scan-plane interlacing.

C:

```
#include "hdf.h"
#include "hcomp.h"

#define WIDTH 5
#define HEIGHT 6
#define PIXEL_DEPTH 3

main( )
{

/* Initialize the image array. */
static uint8 raster_data[HEIGHT][WIDTH][PIXEL_DEPTH] =
{  1, 2, 3,  4, 5, 6,  7, 8, 9,  10,11,12,  13,14,15,
16,17,18, 19,20,21, 22,23,24,  25,26,27,  28,29,30,
31,32,33, 34,35,36, 37,38,39,  40,41,42,  43,44,45,
46,47,48, 49,50,51, 52,53,54,  55,56,57,  58,59,60,
61,62,63, 64,65,66, 67,68,69,  70,71,72,  73,74,75,
76,77,78, 79,80,81, 82,83,84,  85,86,87,  88,89,90 };
intn status;

/* Change interlace from pixel to scan-plane. */
status = DF24setil(DFIL_PLANE);

/* Write the 24-bit image data to file. */
status = DF24addimage("Example2.hdf", (VOIDP)raster_data,
                WIDTH, HEIGHT);

}
```

FORTRAN:

```
      PROGRAM CHANGE INTERLACE

       integer status, d2aimg, d2setil
       integer*4 WIDTH, HEIGHT, PIXEL_DEPTH, DFIL_PLANE
       parameter (WIDTH = 5,
      +    HEIGHT = 6,
      +    PIXEL_DEPTH = 3,
      +    DFIL_PLANE = 2)

       integer raster_data(PIXEL_DEPTH, WIDTH, HEIGHT)

C      Initialize the image array.
```

```
       data raster_data
     + /  1, 2, 3,  4, 5, 6,  7, 8, 9,  10,11,12,  13,14,15,
     +  16,17,18, 19,20,21, 22,23,24,  25,26,27,  28,29,30,
     +  31,32,33, 34,35,36, 37,38,39,  40,41,42,  43,44,45,
     +  46,47,48, 49,50,51, 52,53,54,  55,56,57,  58,59,60,
     +  61,62,63, 64,65,66, 67,68,69,  70,71,72,  73,74,75,
     +  76,77,78, 79,80,81, 82,83,84,  85,86,87,  88,89,90  /

C      Change interlace from pixel to scan plane.
       status = d2setil(DFIL_PLANE)

C      Write the 24-bit raster image to the file.
       status = d2aimg('Example2.hdf', raster_data, WIDTH,
     +                  HEIGHT)

       end
```

7.4.3 Compressing Image Data: DF24setcompress and d2sjpeg

DF24setcompress invokes JPEG compression and sets the JPEG quality and baseline options. To store a 24-bit raster image using JPEG compression, the calling program must contain the following function calls:

C:

```
status = DF24setcompress(type, c_info);
status = DF24addimage(filename, image, width, height);
```

FORTRAN:

```
status = d2scomp(type) OR status = d2sjpeg(quality, baseline)
status = d2aimg(filename, image, width, height, compress)
```

Notice that the calling sequence for C is different from the calling sequence for Fortran-77. Once it is set, the parameter `type` in the **DF24setcompress** routine**,** or **d2scomp** in Fortran-77, routine specifies the compression method that will be used to store the raster images. However, the `c_info` parameter in **DF24setcompress** is missing from **d2scomp** which is a pointer to a structure that contains information specific to the compression method indicated by the `type` parameter. Because data structures of variable size are not supported in Fortran-77, a second compression-specific routine (**d2sjpeg**) is required in the Fortran-77 calling sequence.

For more information about the `c_info` structure refer to Chapter 6, titled *8-bit Raster Images (DFR8 API)*.

Default values for quality and baseline (`quality=75%`, `baseline=on`) are used if `c_info` is a null structure or **d2sjpeg** is omitted. Parameters for **DF24setcompress** and **d24sjpeg** are further described in Table 7E below.

TABLE 7E **DF24setil and DF24setcompress Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DF24setil (d2sil)	il	int32	integer	Interlace format to be set.
DF24setcompress (d2scomp)	type	int32	integer	`COMP_JPEG`.
	c_info	comp_info *	None	Pointer to JPEG information structure.
(d2sjpeg)	quality	None	integer	JPEG compression quality specification.
	baseline	None	integer	JPEG compression baseline specification.

EXAMPLE 3.

Compressing and Writing a 24-Bit Raster Image

In the following examples, **DF24addimage** and **DF24compress** are used to compress a 24-bit image and write it to an HDF file named "Example2.hdf". Notice that compressing an image in C requires only one function call, whereas compressing an image using Fortran-77 requires two. The second Fortran-77 call is required because it is not valid to pass a structure as a parameter in Fortran-77.

C:

```
#include "hdf.h"
#include "hcomp.h"

#define WIDTH 3
#define HEIGHT 5
#define PIXEL_DEPTH 3

main( )
{
/* Initialize the image array. */
static uint8 raster_data[HEIGHT][WIDTH][PIXEL_DEPTH] =
    {  1, 2, 3,  4, 5, 6,  7, 8, 9,
      10,11,12, 13,14,15, 16,17,18,
      19,20,21, 22,23,24, 25,26,27,
      28,29,30, 31,32,33, 34,35,36,
      37,38,39, 40,41,42, 43,44,45 };
static comp_info compress_info;
intn status;

/* Initialize JPEG compression structure. */
compress_info.jpeg.quality = 60;
compress_info.jpeg.force_baseline = 1;

/* Set JPEG compression for storing the image. */
status = DF24setcompress(COMP_JPEG, &compress_info);

/* Write the 24-bit image data to file. */
status = DF24addimage("Example2.hdf", (VOIDP)raster_data,
                WIDTH, HEIGHT);

}
```

FORTRAN:

```
     PROGRAM COMPRESS RIS24

      integer d2aimg, d2scomp, d2sjpeg, status
      integer*4 WIDTH, HEIGHT, PIXEL_DEPTH
      parameter(WIDTH = 3,
     +          HEIGHT = 5,
     +          PIXEL_DEPTH = 3)
      character raster_data(PIXEL_DEPTH, WIDTH, HEIGHT)

C     Initialize the image array.
      data raster_data
     + /  1, 2, 3,  4, 5, 6,  7, 8, 9,
     +   10,11,12, 13,14,15, 16,17,18,
     +   19,20,21, 22,23,24, 25,26,27,
     +   28,29,30, 31,32,33, 34,35,36,
     +   37,38,39, 40,41,42, 43,44,45  /

C     Set compression.
      status = d2scomp(COMP_JPEG)
```

```
C     Set JPEG parameters to quality = 60, and turn compatibility on.
      status = d2sjpeg(60, 1)

C     Write the 24-bit image data to the HDF file.
      status = d2aimg('Example2.hdf', raster_data, WIDTH, HEIGHT)
      end
```

7.5 Reading 24-Bit Raster Images

The DF24 programming model for reading a 24-bit raster image set is as follows:

1. Determine the dimensions for an image if necessary.
2. Specify the interlace format to use when reading the image. (optional)
3. Read the image data from the file.

7.5.1 Reading a Raster Image: DF24getimage

If the dimensions and interlace format of the image are known, **DF24getimage** is the only function call required to read a raster image. If a file is being opened for the first time, **DF24getimage** returns the first image in the file. Additional calls will return successive images in the file, therefore images are read in the same order in which they were written to the file. Normally, **DF24getdims** and **DF24getil** are called before **DF24getimage** so that, if necessary, space allocations and interlace format for the image can be checked and the dimensions verified. If this information is already known, both function calls may be omitted.

The syntax of the **DF24getimage** routine is as follows:

C: `status = DF24getimage(filename, image, width, height);`

FORTRAN: `status = d2gimg(filename, image, width, height)`

DF24getimage retrieves the next 24-bit image from the HDF file specified by the `filename` parameter. If the image is compressed, **DF24getimage** decompresses it and places it in memory at the location pointed to by the `image` parameter. **DF24getimage** assumes the data is stored using pixel interlacing. The space allocated to hold the image is specified by the `width` and `height` parameters and may be larger than the actual image.The parameters for **DF24getimage** are further defined below. (See Table 7F on page 195.)

7.5.2 Determining the Dimensions of an Image: DF24getdims

DF24getdims opens a named file, finds the next image or the first image if the file is being opened for the first time, retrieves the dimensions of the image, then determines the interlace format of the image. Images are read in the order they were written.

To determine the dimensions and interlace format for an image, the calling program must call the following routines:

C:
```
status = DF24getdims(filename, width, height, il);
status = DF24getimage(filename, image, width, height);
```

FORTRAN:
```
status = d2gdim(filename, width, height, il)
status = d2gimg(filename, image, width, height)
```

DF24getdims takes four parameters: `filename`, `width`, `height`, and `il`. It retrieves dimension and interlace format information of the next 24-bit image stored in the HDF file specified by the `filename` parameter. The width and height are returned in the space pointed to by the `width` and `height` parameters respectively. The `il` parameter is used to determine the interlace format. The parameters for **DF24getdims** are further defined below. (See Table 7F.)

7.5.3 Modifying the Interlacing of an Image: DF24reqil

DF24reqil specifies an interlace format to be used when reading a 24-bit image from a file into memory. Regardless of what interlace format is used to store the image, **DF24reqil** forces the image to be loaded into memory using the specified interlace format.

To set or reset the interlace format, the calling program should call the following routines:

C:
```
status = DF24reqil(il);
status = DF24getimage(filename, image, width, height);
```

FORTRAN:
```
status = d2reqil(il)
status = d2gimg(filename, image, width, height)
```

DF24reqil takes `il` as its only parameter. Valid `il` values are `DFIL_PIXEL`, `DFIL_LINE` and `DFIL_PLANE`. As a call to **DF24reqil** may require a substantial reordering of the data, a much slower I/O performance than would be achieved if the interlace format wasn't reset may result.

The parameters of **DF24reqil** is further defined below. (See Table 7F.)

TABLE 7F **DF24getimage, DF24getdims and DF24reqil Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DF24getimage (d2gimg)	filename	char *	character* (*)	Name of the HDF file containing the raster image.
	image	VOIDP	<valid numeric data type>	Buffer for the raster image.
	width	int32	integer	Width of the raster image buffer.
	height	int32	integer	Height of the raster image buffer.
DF24getdims (d2gdims)	filename	char *	character* (*)	Name of HDF file containing the raster image.
	width	int32 *	integer	Pointer to the number of columns in the raster image.
	height	int32 *	integer	Pointer to the number of rows in the raster image.
	il	intn	integer	Pointer to the interlace format of the raster image.
DF24reqil (d2reqil)	il	intn	integer	Pointer to the interlace format of the raster image.

EXAMPLE 4. **Reading a 24-Bit Raster Image from an HDF File**

The following examples read a 24-bit image from the "Example2.hdf" HDF file created in Example 2. Although the **DF24getdims** function call is optional, it is included as a demonstration of how to verify the image dimensions and interlace format before reading the image data. If the image dimensions and interlace format are known, only the **DF24getimage** call is required.

C:

```
#include "hdf.h"

#define WIDTH 5
#define HEIGHT 6
#define PIXEL_DEPTH 3

main( )
{
uint8 raster_data[PIXEL_DEPTH][HEIGHT][WIDTH];
int32 width, height;
intn interlace, status;

/* Get the image dimensions from the HDF file. */
status = DF24getdims("Example2.hdf", &width, &height,
                &interlace);

/*
* Read raster data if the dimensions are
* correct.
*/
if (width <= WIDTH && height <= HEIGHT)
  status = DF24getimage("Example2.hdf", (VOIDP)raster_data,
                width, height);
}
```

FORTRAN:

```
     PROGRAM READ RIS24

      integer d2gimg, d2gdims, status, width, height, interlace
      integer*4 X_LENGTH, Y_LENGTH, PIXEL_DEPTH
      parameter(X_LENGTH = 5, Y_LENGTH = 6, PIXEL_DEPTH = 3)
      integer raster_data(PIXEL_DEPTH, X_LENGTH, Y_LENGTH)

C     Read the dimensions raster image.
      status = d2gdims('Example2.hdf', width, height, interlace)

C     Read image data from the HDF file if the dimensions are
C     correct.
      if (width .eq. X_LENGTH .and. height .eq. Y_LENGTH) then
       status = d2gimg('Example2.hdf', raster_data, width, height)
      endif

      end
```

7.5.4 Reading a 24-Bit Raster Image with a Given Reference Number: DF24readref

DF24readref is used to access specific images stored in files containing multiple raster image sets. It is optionally used before **DF24getimage**. **DF24readref** can be used in connection with vgroups, which identify their members by tag/reference number pairs. See Chapter 5, titled *Vgroups (V API)*, for a discussion of vgroups and tag/reference number pairs.

To access a specific raster image set, use the following sequence of routine calls:

C:

```
status = DF24readref(filename, ref);
status DF24getimage(filename, image, width, height);
```

FORTRAN:

```
status = d2rref(filename, ref)
status = d2gimg(filename, image, width, height)
```

DF24readref sets the reference number for the next read operation performed on the HDF file `filename` to the reference number contained in `ref`. Because reference numbers are not always assigned in sequence, it is not guaranteed that a reference number represents the location of the image in the file.

The parameters of **DF24readref** are further described in the following table.

TABLE 7G **DF24readref Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DF24readref (**d2rref**)	filename	char *	character* (*)	Name of HDF file containing the raster image.
	ref	uint16	integer	Reference number for the next call to **DF24getimage**.

7.5.5 Specifying that the Next Image Read to be the First 24-Bit Raster Image in the File: DF24restart

DF24restart causes the next call to **DF24getimage** or **DF24getdims** to read from the first raster image set in the file, rather than the RIS24 following the one that was most recently read. Use the following call to invoke **DF24restart**:

C: `status = DF24restart( );`

FORTRAN: `status = d2first( )`

TABLE 7H **DF24restart Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DF24restart (**d2first**)	None	None	None	None.

7.6 24-Bit Raster Image Information Retrieval Routines

7.6.1 Querying the Total Number of Images in a File: DF24nimages

DF24nimages returns the total number of 24-bit raster image sets in a file, and has the following syntax:

C: `num_of_images = DF24nimages(filename);`

FORTRAN: `num_of_images = d2nimg(filename)`

TABLE 7I

DF24nimages Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DF24nimages (d2nimg)	filename	char *	character* (*)	Name of the HDF file.

7.6.2 Querying the Reference Number of the Most Recently Read or Written 24-Bit Raster Image: DF24lastref

DF24lastref returns the reference number of the 24-bit raster image most recently read or written. This routine is used for attaching annotations to images and adding images to vgroups. (See Chapter 5, titled *Vgroups (V API)* and Chapter 10, titled *Annotations (DFAN API)* for details on how to use reference numbers in connection with these applications.

The following calling sequence uses **DF24lastref** to find the reference number of the RIS24 most recently added to an HDF file:

C:

```
status = DF24addimage(filename, image, width, height, compress);
lastref = DF24lastref( );
```

FORTRAN:

```
status = d2aimg(filename, image, width, height, compress)
lastref = d2lref( )
```

DF24putimage or **DF24getimage** can be used in place of **DF24addimage** with similar results.

TABLE 7J

DF24lastref Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DF24lastref (d2lref)	filename	None	None	None.

Chapter 8

General Raster Images (GR API)

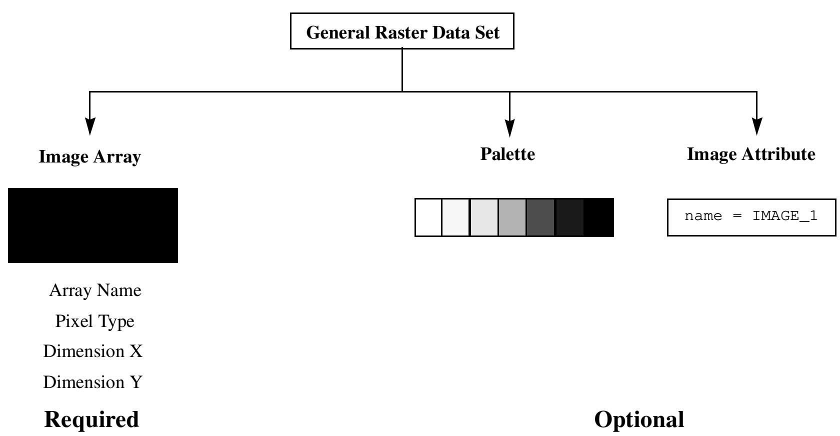
8.1 Chapter Overview

This chapter describes the general raster (GR) data model and the interface routines used to manipulate general raster data objects. As the general raster data model is designed to provide a more flexible means of storing raster image data than the RIS8 and RIS24 models, a brief comparison of the GR model with the RIS8 and RIS24 data models is also provided.

8.2 The General Raster Data Model

HDF users familiar with the SD interface will find the general raster data model a simplified version of the SD scientific data set model, customized to accommodate image data storage and manipulation. The raster image data is stored in a two-dimensional array and attributes can be created for the image, the file or both. Palettes can be created and attached to the image as well as compression method information. A fundamental difference between the SD scientific data model and the GR raster data model is the absence of customizable dimensional information in the GR data set. Refer to Figure 8a.

FIGURE 8a **General Raster Image Data Set Contents**



8.2.1 Required General Raster Objects

Every general raster data set must contain three objects. These objects are the ***image array*** , the ***image array name*** and ***pixel type*** . Required objects are automatically created from the information provided at the time the data set is defined.

8.2.1.1 Image Array

An ***image array*** is the two-dimensional structure used to store the image's pixel data.

8.2.1.2 Image Array Name

Each image array has an ***image name*** consisting of a string of case-sensitive alphanumeric characters. The name must be provided by the calling program; the GR interface does not provide one by default if one isn't specified. Names are assigned when the data set is created and cannot be changed afterwards. Image array names do not have to be unique within a file, but if they are not it can be difficult to distinguish between the general raster data sets in the file.

8.2.1.3 Pixel Type

Another fundamental difference between the SD SDS model and the GR data model is that a GR image data array is defined by data type of its elements and the number of components in each element rather than only the data type. This is because each element in an image array corresponds to one pixel and each element of pixel data can consist of a variable number of color component values. (Red-Blue-Green or RGB, Cyan-Magenta-Yellow-Black or CMYK, etc.) These color component values can be represented by different methods and data lengths (8-bit lookup table or 24-bit direct representation, graphically depicted in Figure 6a in Chapter 6 and Figure 7b in Chapter 7 respectively). The data type of array elements and the number of components in each element are collectively referred to as the ***pixel type***.

Data Type

As of HDF version 4.1r1, the general raster data model supports any data type.

Number of Components

Pixel elements can be comprised of any number of components.

8.2.1.4 Dimensions

Image array ***dimensions*** specify the size of an image array. There are no routines in the GR library that allow the HDF user to add attributes to a dimension or change the scale, as is possible in the SD interface.

8.2.2 Optional General Raster Objects

There are two types of optional objects available for inclusion in a general raster data set: ***palettes*** and ***attributes***. These objects are only created when specifically requested by the calling program; the GR interface doesn't provide predefined palettes or attributes.

Palettes

Palettes are lookup tables attached to images for the purpose of the determining a set of color values for each pixel value in the image array. The GR interface provides similar capabilities for storing and manipulation palettes as the DFP interface described in Chapter 9, titled *Palettes (DFP API)*. However, the DFP interface is restricted to single-file operations while the GR interface allows multifile palette operations. Eventually, all palette manipulation functionality will reside only within the GR interface, but for the time being the single-file DFP routines are fully compatible with palettes created with the GR palette routines. The GR palette routines are described in Section 8.9 on page 233.

Attributes

Attributes are defined by the calling program to contain auxiliary information about a file, image or both. They are described in Chapter 3, titled *Scientific Data Sets (SD API)*. The GR implementation of attributes is covered in Section 8.8 on page 227.

8.3 The General Raster API

The GR interface consists of routines for storing, retrieving and manipulating the data in general raster data sets.

8.3.1 GR API Routines

All C routine names in the general raster interface have the prefix “GR” and the equivalent Fortran-77 routine names are prefaced by "mg". All GR routines are classifiable within one of the following categories:

- ***Access routines*** initialize and terminate access to the GR interface and general raster data sets.
- ***Read/write routines*** modify the data and metadata contained in a general raster data set.
- ***Maintenance routines*** create and destroy the data and metadata contained in a general raster data set and modify global settings governing the format of the stored data.
- ***Inquiry routines*** return information about data contained in a general raster data set.

The GR function calls are listed in the following table and described further in the *HDF Reference Guide*.

TABLE 8A **GR Library Routines**

Purpose	Routine Name		Description
	C	Fortran-77	
Access	GRstart	mgstart	Initializes the GR interface for a given data file.
	GRend	mgend	Terminates access to the file initialized by **GRstart**.
	GRselect	mgselct	Selects the data set to perform operations on.
	GRendaccess	mgendac	Terminates access to the data set selected by **GRselect** or **GRcreate**.
Read/write	GRreadimage	mgrdimg/ mgrcimg	Reads image data from a general raster data set.
	GRwriteimage	mgwrimg/ mgwcimg	Writes image data to a general raster data set.
	GRidtoref	mgid2ref	Maps an general raster data set identifier to a reference number.
	GRluttoref	None	Returns the reference number for the specified palette..
	GRreftoindex	mgr2idx	Maps the reference number of a general raster data set to a data set index and returns the index.
	GRnametoindex	mgn2ndx	Maps a raster image name to an index and returns the index.
	GRreadlut	mgrdlut/ mgrclut	Reads palette data from a general raster data set.
	GRwritelut	mgwrlut/ mgwclut	Writes palette data to a general raster data set.
	GRsetattr	mgsnatt/ mgscatt	Writes the attribute of an object to a general raster data set.
	GRgetattr	mggnatt/ mggcatt	Reads the attribute of an object from a general raster data set.

Maintenance	GRcreate	mgcreat	Creates a new general raster data set.
	GRreqlutil	mgrltil	Sets the interlace mode for the next palette read from a general raster data set.
	GRreqimageil	mgrimil	Sets the interlace mode for the next image read from a general raster data set.
	GRgetlutid	mggltid	Allocates a palette id to a general raster data set.
	GRsetexternalfile	mgsxfil	Specifies that the image data of a general raster data set is a special element of an external element.
	GRsetaccesstype	mgsactp	Sets the access to a general raster data set to be either parallel or serial.
	GRsetcompress	mgscomp	Makes the image data of a general raster data set a compressed special element.
Inquiry	GRfileinfo	mgfinfo	Returns global information on the specified GR data set.
	GRgetiminfo	mggiinf	Returns information on the specified general raster data set.
	GRgetlutinfo	mgglinf	Returns information on a given palette.
	GRattrinfo	mgatinf	Returns attribute information on a given object.
	GRfindattr	mgfndat	Returns the index of an attribute with the given name.

8.4 Programming Model for the GR Interface

As with the SD interface, the GR interface relies on the calling program to initiate and terminate access to files and data sets to support multifile access. The GR programming model for accessing a general raster data set is as follows:

1. Open a file by obtaining a file id from a file name.
2. Initialize the GR interface and obtain a general raster interface id.
3. Open an existing general raster data set by obtaining a data set id from the interface id and data set index **OR** create a new general raster data set by obtaining a data set id from the data set name and data type.
4. Perform desired operations on the data set.
5. Dispose of the raster image id.
6. Terminate access to the GR interface by disposing of the interface id.
7. Close the file.

To access a single raster image data set in an HDF file, the calling program must contain the following calls:

C:

```
file_id = Hopen(filename, access_mode, number_of_desc);
gr_id = GRstart(file_id);
ri_id = GRselect(gr_id, gr_index);
<Optional operations>
status = GRendaccess(ri_id);
status = GRend(gr_id);
status = Hclose(file_id);
```

FORTRAN:

```
file_id = hopen(filename, access_mode, number_of_desc)
gr_id = mgstart(file_id)
ri_id = mgselct(gr_id, gr_index)
<Optional operations>
status = mgendac(ri_id)
status = mgend(gr_id)
status = hclose(file_id)
```

To access several files at the same time, a calling program must obtain a separate file id for each file to be opened. Similarly, to access more than one general raster data set a calling program must obtain a separate interface id for each data set. For example, to open two data sets stored in two files a program would execute the following series of function calls:

C:

```
file_id_1 = Hopen(filename_1, access_mode, number_of_desc);
gr_id_1 = GRstart(file_id_1);
ri_id_1 = GRselect(gr_id_1, gr_index_1);
file_id_2 = Hopen(filename_2, access_mode, number_of_desc);
gr_id_2 = GRstart(file_id_2);
ri_id_2 = GRselect(gr_id_2, gr_index_2);
<Optional operations>
status = GRendaccess(ri_id_1);
status = GRend(gr_id_1);
status = Hclose(file_id_1);
status = GRendaccess(ri_id_2);
status = GRend(gr_id_2);
status = Hclose(file_id_2);
```

FORTRAN:

```
file_id_1 = hopen(filename_1, access_mode, number_of_desc)
gr_id_1 = mgstart(file_id_1)
ri_id_1 = mgselct(gr_id_1, gr_index_1)
file_id_2 = hopen(filename_2, access_mode, number_of_desc)
gr_id_2 = mgstart(file_id_2)
ri_id_2 = mgselct(gr_id_2, gr_index_2)
<Optional operations>
status = mgendac(ri_id_1)
status = mgend(gr_id_1)
status = hclose(file_id_1)
status = mgendac(ri_id_2)
status = mgend(gr_id_2)
status = hclose(file_id_2)
```

Because every file and general raster data set is assigned its own identifier, the order in which files and data sets are accessed is very flexible as long as all file and general raster data set ids are individually discarded before the end of the calling program.

8.4.1 Accessing Files and Images: GRstart and GRselect

In the GR interface, **Hopen** is used to open files. This differs from the SD programming model where **SDstart** is used for this purpose. For information on the use of **Hopen** refer to Chapter 2, titled *HDF Fundamentals* and for information on **SDstart** refer to Chapter 3, titled *Scientific Data Sets (SD API)*.

GRstart initializes the GR interface and must be called once after **Hopen** and before any other GR routines are called. It takes one argument; the `file_id` returned by **Hopen** and returns the interface id `gr_id`.

GRselect specifies the given image as the current image to be accessed. It takes two arguments: the `gr_id` returned by **GRstart** and `gr_index` and returns the raster image id `ri_id`. The argument `gr_index` is the position of the data set relative to the beginning of the file. The argument `gr_index` is zero-based, meaning that the index of first image in the file is 0.

The parameters for **GRstart** and **GRselect** are further defined below. (See Table 8B.)

8.4.2 Terminating Access to Files and Images: GRendaccess and GRend

GRendaccess disposes of the open raster image id `ri_id` and terminates access to the data set initiated by the corresponding call to **GRselect**. The calling program must make one **GRendaccess** call for every **GRselect** call made during its execution. Failing to call **GRendaccess** for each call to **GRselect** may result in a loss of data.

GRend disposes of the general raster data set id `gr_id` and terminates the access to the GR interface initiated by the corresponding call to **GRstart**. The calling program must make one **GRend** call for every **GRstart** call made during its execution and failing to call **GRend** for each **GRstart** may result in a loss of data. The parameters for **GRendaccess** and **GRend** are listed and defined in the following table.

TABLE 8B **GRstart, GRselect, GRend and GRendaccess Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type C	Data Type Fortran-77	Description
GRstart (mgstart)	file_id	int32	integer	File identifier.
GRselect (mgselct)	gr_id	int32	integer	General raster data set identifier.
	gr_index	int32	integer	Position of the general raster data set within the file.
GRend (mgend)	gr_id	int32	integer	General raster data set identifier.
GRendaccess (mgendac)	ri_id	int32	integer	Image array identifier.

8.5 Creating and Writing General Raster Images

This section describes the routines needed to create and write simple general raster data sets. A "simple" general raster data set is defined here as one with no attributes.

Creating a general raster data set and writing data to it are separate operations in the GR programming model. Once an array is defined it is ready to store data. No definitions are retained about the size, contents or number of components of a general raster data set from one data set to the next or from one file to the next.

8.5.1 Creating General Raster Images: GRcreate

Creating a simple general raster data set requires the following steps:

1. Open a file and initialize the GR interface.
2. Define the characteristics of the general raster data set.
3. Perform optional operations on the data set.
4. Terminate access to the data set.
5. Terminate access to the GR interface and close the file.

To create the data set, the calling program must contain the following sequence of function calls:

C:

```
file_id = Hopen(filename, access_mode, number_of_desc);
gr_id = GRstart(file_id);
ri_id = GRcreate(gr_id, name, ncomp, number_type, il,
        dim_sizes);
<Optional operations>
status = GRendaccess(ri_id);
status = GRend(gr_id);
status = Hclose(file_id);
```

FORTRAN:

```
file_id = hopen(filename, access_mode, number_of_desc)
gr_id = mgstart(filename, access_mode)
ri_id = mgcreat(gr_id, name, ncomp, number_type, il,
        dim_sizes);
<Optional operations>
```

```
status = mgendac(ri_id)
status = mgend(gr_id)
status = hclose(file_id)
```

GRcreate defines a new general raster data set using the arguments `name`, `ncomp`, `number_type`, `il` and `dim_sizes`. Once a data set is created, you cannot change its name, data type, dimension or number of components. **GRcreate** does not actually perform the write; this occurs only when **GRendaccess** is called.

The maximum length of a data set name is defined by `MAX_GR_NAME` and the maximum number of components of an image array is defined by `MAX_VAR_DIMS`. Both are defined in the "hlimits.h" header file.

When creating a general raster data set, it is necessary to specify the data type of the image array data. The "hntdefs.h" header file contains definitions of all valid data types, which are described in Chapter 2, titled *HDF Fundamentals*. The GR interface supports all regular HDF data types. However, it does not include the support for "n-bit" integers which is included in the SD interface.

TABLE 8C **GRcreate Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
GRcreate (mgcreat)	gr_id	int32	integer	General raster data set identifier.
	name	char *	character* (*)	Name of the image.
	ncomp	int32	integer	Number of components in each element of the data set.
	data_type	int32	integer	Data type of the data set.
	interlace_mode	int32	integer	Interlace mode to be used when writing to the data set
	dim_sizes	int32 [2]	integer (2)	Array defining the size of both dimensions.

8.5.2 Setting the Interlace Mode

During the process of reading a palette or an image, the interlace mode can be set to either MFGR_INTERLACE_PIXEL, MFGR_INTERLACE_LINE or MFGR_INTERLACE_COMPONENT. These definitions respectively correspond to pixel interlacing, line interlacing and component interlacing. The first two interlacing modes are illustrated for the instance of 24-bit pixel representation in Figure 7c of Chapter 7, titled *24-bit Raster Images (DF24 API)*. Component interlacing, as the name implies, describes interlacing raster data by color component.

EXAMPLE 1. **Creating and Accessing a General Raster Data Set**

In these examples an empty array is created, or one that has been defined but not yet initialized with data.

C:

```
#include “hdf.h”
#include “mfgr.h”

#define X_LENGTH 5
#define Y_LENGTH 10

main( )
{

int32 gr_id, ri_id, file_id, status;
int32 dimsizes[2], ncomp, il;

/* Create and open the file. */
file_id = Hopen(“Example1.hdf”, DFACC_CREATE, 0);

/* Initiate the GR interface. */
gr_id = GRstart(file_id);

/* Define the number of components and dimensions of the image. */
ncomp = 2;
il = MFGR_INTERLACE_PIXEL;
dimsizes[0] = X_LENGTH;
dimsizes[1] = Y_LENGTH;

/* Create the image array. */
ri_id = GRcreate(gr_id, “Image_array_1”, ncomp, DFNT_INT16, il,
                dimsizes);

/* Terminate access to the image array. */
status = GRendaccess(gr_id);

/* Terminate access to the GR interface and close the file */
status = GRend(ri_id);

/* Close the file. */
status = Hclose(file_id);

}
```

FORTRAN:

```
     PROGRAM CREATE IMAGE ARRAY

      integer*4 gr_id, ri_id, file_id, dimsizes(2), ncomp, il
      integer mgstart, mgcreat, mgendac, mgend, hopen, hclose
      integer*4 X_LENGTH, Y_LENGTH, status
      parameter (X_LENGTH = 5, Y_LENGTH = 10,
     +           MFGR_INTERLACE_PIXEL = 0)
C     DFACC_CREATE and DFNT_INT16 are defined in hdf.h.
      integer*4 DFACC_CREATE, DFNT_INT16
      parameter (DFACC_CREATE = 4, DFNT_INT16 = 22)

C     Create and open the file.
      file_id = hopen(“Example1.hdf”, DFACC_CREATE, 0)

C     Initiate the GR interface.
      gr_id = mgstart(file_id)

C     Define the number of components and dimensions of the image
C     array.
```

```
      ncomp = 2
      il = MFGR_INTERLACE_PIXEL
      dimsizes(1) = Y_LENGTH
      dimsizes(2) = X_LENGTH

C     Create the image array.
      ri_id = mgcreat(gr_id, 'Ex_array_1', ncomp, DFNT_INT16, il,
     +                dimsizes)

C     Terminate access to the image array.
      status = mgendac(ri_id)

C     Terminate access to the GR interface and close the file.
      status = mgend(gr_id)

C     Close the file.
      status = hclose(file_id)
      end
```

8.5.3 Writing General Raster Images: GRwriteimage

GRwriteimage is the routine used to either completely or partially fill an n-component image array. It can also skip a specified number of image array elements between write operations along each dimension.

Writing data to an image array involves the following steps:

1. Open a file and initialize the GR interface.
2. Select or create a data set.
3. Write data to the image array.
4. Terminate access to the data set.
5. Terminate access to the file and close the file.

The calling program must contain the following sequence of calls:

C:

```
file_id = Hopen(filename, access_mode, number_of_desc);
gr_id = GRstart(file_id);
ri_id = GRcreate(gr_id, name, ncomp, number_type, il,
dim_sizes);
status = GRwriteimage(ri_id, start, stride, edge, data);
status = GRendaccess(gr_id);
status = GRend(ri_id);
status = Hclose(file_id);
```

FORTRAN:

```
file_id = hopen(filename, access_mode, number_of_desc)
gr_id = mgstart(file_id)
ri_id = mgcreat(gr_id, name, ncomp, number_type, il,
dim_sizes);
status = mgwrimg(ri_id, start, stride, edge, data)
status = mgendac(ri_id)
status = mgend(gr_id)
status = hclose(file_id)
```

As with SD arrays, subsets of general raster images can be written. (In n-dimensional SD arrays these two-dimensional subsets are referred to as “slabs”.) A subset is specified by giving an array consisting of the coordinate location of the origin vertex and an array consisting of lengths of each of the two dimensions.

GRwriteimage takes five arguments: `ri_id`, `start`, `stride`, `edge`, and `data`. The `ri_id` argument is the raster image id returned by **GRcreate** or **GRselect**. The arguments `start`, `stride`, and `edge` respectively describe the initial location in the image for the write operation, the number of locations the current array location will be moved forward after each write, and the length of each dimension of the subset to be written. If the image array is smaller than the `data` argument array, the amount of data written will be truncated to the size of the image array. These concepts are explained further in Chapter 3, titled *Scientific Data Sets (SD API)*.

There are two Fortran-77 versions of this routine: **mgwrimg** and **mgwcimg**. The **mgwrimg** routine writes buffered numeric data and the **mgwcimg** routine writes buffered character data.

The parameters for **GRwriteimage** are described in the following table. Note that, because there are two Fortran-77 versions of **GRwriteimage**, there are correspondingly two entries in the "Data Type" field of the *data* parameter.

TABLE 8D

GRwriteimage Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
GRwriteimage (mgwrimg/ mgwcimg)	ri_id	int32	integer	Image array identifier returned by **GRcreate**.
	start	int32 [2]	integer (2)	Array containing the x,y-coordinate location the write will start for each dimension.
	stride	int32 [2]	integer (2)	Array containing the number of data locations the current location is to be moved forward before the next write.
	edge	int32 [2]	integer (2)	Array containing the number of data elements that will be written along each dimension.
	data	VOIDP	<valid numeric data type>	Buffer array containing the data to be written.

EXAMPLE 2.

Creating and Writing an Image

These examples use **GRcreate** and **GRwriteimage** under the C interface and **mgcreat** and **mgwrimg** under the Fortran-77 interface.

C:

```
#include "hdf.h"
#include "mfgr.h"

#define X_LENGTH 15
#define Y_LENGTH 10

main( )
{
int32 gr_id, ri_id, file_id, status;
int32 dims[2], start[2], edges[2], ncomp, il;
int16 image_data[Y_LENGTH][X_LENGTH][2], i, j;

/* Create and open the file. */
file_id = Hopen("Example2.hdf", DFACC_CREATE, 0);

/* Initiate the GR interface. */
gr_id = GRstart(file_id);
```

```
/* Define the number of components and dimensions of the image. */
ncomp = 2;
il = MFGR_INTERLACE_PIXEL;
dims[0] = X_LENGTH;
dims[1] = Y_LENGTH;

/* Create the array. */
ri_id = GRcreate(gr_id, "Ex_array_2", ncomp, DFNT_INT16, il, dims);

/* Fill the stored-data array with values. */
for (j = 0; j < Y_LENGTH; j++) {
    for (i = 0; i < X_LENGTH; i++) {
        image_data[j][i][0] = (i + j) + 1;
        image_data[j][i][1] = (i + j) + 1;
    }
    }

/* Define the location, pattern, and size of the data set */
for (i = 0; i < 2; i++) {
    start[i] = 0;
    edges[i] = dims[i];
}

/* Write the stored data to the image array. */
status = GRwriteimage(ri_id, start, NULL, edges, (VOIDP)image_data);

/* Terminate access to the array. */
status = GRendaccess(ri_id);

/* Terminate access to the GR interface. */
status = GRend(gr_id);

/* Close the file. */
status = Hclose(file_id);

}
```

FORTRAN:

```
     PROGRAM WRITE ARRAY

      integer*4 ri_id, gr_id, file_id, ncomp, il
      integer dims(2), start(2), edges(2), stride(2)
      integer i, j, status
      integer mgstart, mgcreat, mgwrimg, mgendac, mgend
      integer hopen, hclose
      integer*4 DFACC_CREATE, DFNT_INT16
      integer*4 X_LENGTH, Y_LENGTH
      parameter (DFACC_CREATE = 4, DFNT_INT16 = 22, X_LENGTH = 15,
     +           Y_LENGTH = 10, MFGR_INTERLACE_PIXEL = 0)
      integer*2 image_data(2, X_LENGTH, Y_LENGTH)

C     Create and open the file.
      file_id = hopen('Example2.hdf', DFACC_CREATE, 0)

C     Initiate the GR interface.
      gr_id = mgstart(file_id)

C     Define the number of components and dimensions of the image.
      ncomp = 2
      il = MFGR_INTERLACE_PIXEL
```

```
      dims(1) = X_LENGTH
      dims(2) = Y_LENGTH

C     Create the data set.
      ri_id = mgcreat(gr_id, 'Ex_array_2', ncomp, DFNT_INT16, il, dims)

C     Fill the stored-data array with values.
      do 20 j = 1, Y_LENGTH
         do 10 i = 1, X_LENGTH
            image_data(1, i, j) = i + j - 1
            image_data(2, i, j) = i + j - 1
10       continue
20    continue

C     Define the location, pattern, and size of the data set
C     that will be written to.
      start(1) = 0
      start(2) = 0
      edges(1) = X_LENGTH
      edges(2) = Y_LENGTH
      stride(1) = 1
      stride(2) = 1

C     Write the stored data to the image array.
      status = mgwrimg(ri_id, start, stride, edges, image_data)

C     Terminate access to the array.
      status = mgendac(ri_id)

C     Terminate access to the GR interface.
      status = mgend(gr_id)

C     Close the file.
      status = hclose(file_id)

      end
```

8.6 Reading Data from an Image: GRreadimage

Reading one or more slabs from an image array involves the following steps:

1. Open the file and initialize the GR interface.
2. Select a data set.
3. Read data from the image array.
4. Terminate access to the data set.
5. Terminate access to the GR interface and close the file.

To read data from an image array, the calling program must contain the following function calls:

C:

```
file_id = Hopen(filename, access_mode, number_of_desc);
gr_id = GRstart(file_id);
ri_id = GRselect(gr_id, gr_index);
status = GRreadimage(ri_id, start, stride, edge, data);
status = GRendaccess(gr_id);
status = GRend(ri_id);
status = Hclose(file_id);
```

FORTRAN:

```
file_id = hopen(filename, access_mode, number_of_desc)
```

```
gr_id = mgstart(file_id)
ri_id = mgselct(gr_id, GRs_index)
status = mgrdimg(ri_id, start, stride, edge, data)
status = mgendac(gr_id)
status = mgend(ri_id)
status = hclose(file_id)
```

GRreadimage can be used to read either an entire image or a subset of the image. The `ri_id` argument is the raster image identifier returned by **GRselect**. As with **GRwriteimage**, the arguments `start`, `stride`, and `edge` respectively describe the starting location for the read operation, the number of locations the current image array location will be moved forward after each read, and the length of each dimension to be read. If the image array is smaller than the `data` argument array, the amount of data read will be limited to the maximum size of the image array.

There are two Fortran-77 versions of this routine: **mgrdimg** and **mgrcimg**. The **mgrdimg** routine reads numeric image data and **mgrcimg** reads character image data.

The parameters for **GRreadimage** are further defined below. (See Table 8E.)

8.6.1 Reading General Raster Images from an External File

Image data is read from an external or compressed file in the same way that it is read from a primary file.

8.6.2 Setting the Interlace Mode for a Palette or Image Read

There are two GR routines that deal with setting the interlace in raster image data sets: **GRreqlutil** and **GRreqimageil**. Refer to Section 8.5.2 on page 205 for a description of the GR interlace modes.

The **GRreqlutil** routine sets the interlace mode for the next palette read. It can be called at anytime before the read operation and takes two parameters, `ri_id` and `interlace_mode`. The `ri_id` parameter is the image array identifier returned by the **GRselect** routine and the `interlace_mode` is the interlace mode that will be in effect for the next image or palette read operation.

The **GRreqimageil** routine sets the interlace mode for the next image read. It also can be called at anytime before the read operation and takes the same two parameters as **GRreqlutil**. **GRreqlutil** and **GRreqimageil** may be called more than once; the interlace mode setting specified by the last call to either routine will be used for the next read operation.

Descriptions of both interlace mode routines are included in the following table. The GR palette read and write routines are covered in Section 8.9 on page 233. Note that, because there are two Fortran-77 versions of **GRreadimage**, there are correspondingly two entries in the “Data Type” field of the *data* parameter.

TABLE 8E **GRreadimage, GRreqlutil and GRreqimageil Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	

GRreadimage (mgrdimg/ mgrcimg)	ri_id	int32	integer	Image identifier.
	start	int32[2]	integer (2)	Array containing the starting read coordinates.
	stride	int32[2]	integer (2)	Array containing the number of data locations the current location is to be moved forward before the next read.
	edge	int32[2]	integer (2)	Array containing the number of data element that will be read along each dimension.
	data	VOIDP	<valid numeric data type>	Buffer for the returned image data.
GRreqlutil (mgrltil)	ri_id	int32	integer	Image identifier.
	interlace_mode	intn	integer	Interlace mode for the next palette read operation.
GRreqimageil (mgrimil)	ri_id	int32	integer	Image identifier
	interlace_mode	intn	integer	Interlace mode for the next image read operation.

EXAMPLE 3.

Reading an Image

When **GRreadimage** is used to read an entire image array, the coordinates for the start position must begin at 0 for each dimension (`start={0,0}`), the interval between each read must equal 1 for each dimension (`stride=NULL or stride={1,1}`), and the size of each dimension must equal the size of the array itself (`edge={dim_size_1, dim_size_2}`). The `data` buffer must have enough space allocated to hold the data. In these examples, the image created in Example 2 is read.

C:

```
#include "hdf.h"

#define X_LENGTH 15
#define Y_LENGTH 10

main( )
{
int32 gr_id, ri_id, file_id, status;
int32 start[2], edges[2], dims[2], nattrs;
int16 image_data[Y_LENGTH][X_LENGTH][2];

/* Open the file. */
file_id = Hopen("Example2.hdf", DFACC_RDONLY, 0);

/* Open the file and initiate the GR interface. */
gr_id = GRstart(file_id);

/* Select the first (and in this case, only) data set in the file. */
ri_id = GRselect(gr_id, 0);

/* Define the location, pattern, and size of the data to read. */
dims[0] = X_LENGTH;
dims[1] = Y_LENGTH;
start[0] = start[1] = 0;
edges[0] = dims[0];
edges[1] = dims[1];

/* Read the data in the image array. */
status = GRreadimage(ri_id, start, NULL, edges, (VOIDP)image_data);

/* Terminate access to the image array */
status = GRendaccess(ri_id);
```

```
/* Terminate access to the GR interface. */
status = GRend(gr_id);

/* Close the file. */
status = Hclose(file_id);

}
```

FORTRAN:

```
      PROGRAM READ ARRAY

       integer*4 gr_id, ri_id, file_id, status
       integer start(2), edge(2), stride(2)
       integer hopen, hclose, mgstart, mgselct, mgrdimg
       integer mgendac, mgend

C      DFACC_RDONLY is defined in hdf.h. MAX_NC_NAME and MAX_VAR_DIMS
C      are defined in netcdf.h.
       integer*4 DFACC_RDONLY, MAX_NC_NAME, MAX_VAR_DIMS
       integer*4 X_LENGTH, Y_LENGTH
       parameter (DFACC_RDONLY = 1, MAX_NC_NAME = 256,
      +           MAX_VAR_DIMS = 32, X_LENGTH = 15,
      +           Y_LENGTH = 10)
       integer*2 array_data(2, X_LENGTH, Y_LENGTH)
       integer dims(MAX_VAR_DIMS)

C      Open the file.
       file_id = hopen('Example2.hdf', DFACC_RDONLY, 0)

C      Initiate the GR interface.
       gr_id = mgstart(file_id)

C      Select the first (and in this case, only) data set in the file.
       ri_id = mgselct(gr_id, 0)

C      Define the location, pattern, and size of the data to read
C      from the data set.
       dims(1) = X_LENGTH
       dims(2) = Y_LENGTH
       start(1) = 0
       start(2) = 0
       stride(1) = 1
       stride(2) = 1
       edge(1) = dims(1)
       edge(2) = dims(2)

C      Read the array data set.
       status = mgrdimg(ri_id, start, stride, edge, array_data)

C      Terminate access to the image array.
       status = mgendac(ri_id)

C      Terminate access to the GR interface and the file.
       status = mgend(gr_id)

C      Close the file.
       status = hclose(file_id)

       end
```

EXAMPLE 4.

Reading a Subset of an Image Array

The following examples show how the `start` and `edges` arguments of the **GRreadimage** function can be initialized so that a subset of the data set can be read.

C:

```
#include “hdf.h”

#define X_LENGTH 15
#define Y_LENGTH 10

main( )
{
int32 gr_id, ri_id, file_id;
int32 start[2], edges[2];
int16 all_image[2][Y_LENGTH][X_LENGTH];
int16 subset_image[2][7][3], status;

/* Open the file. */
file_id = Hopen(“Example2.hdf”, DFACC_RDONLY, 0);

/* Open the file for read-only access. */
gr_id = GRstart(file_id);

/* Select the first data set. */
ri_id = GRselect(gr_id, 0);

/* First, read the entire data set. */
start[0] = start[1] = 0;
edges[0] = X_LENGTH;
edges[1] = Y_LENGTH;
status = GRreadimage(ri_id, start, NULL, edges, (VOIDP)all_image);

/* Read a subset of the data set. */
start[0] = 1;
start[1] = 1;
edges[0] = 3;
edges[1] = 7;
status = GRreadimage(ri_id, start, NULL, edges, (VOIDP)subset_image);

/* Terminate access to the array. */
status = GRendaccess(ri_id);

/* Terminate access to the GR interface. */
status = GRend(gr_id);

/* Close the file. */
status = Hclose(file_id);

}
```

FORTRAN:

```
      PROGRAM READ SUBSET

       integer*4 gr_id, ri_id, file_id
       integer start(2), edges(2), stride(2), status
       integer mgstart, mgselct, mgrdimg, mgendac, mgend
       integer hopen, hclose
```

```
C     DFACC_RDONLY is defined in hdf.h.
      integer*4 DFACC_RDONLY
      integer*4 X_LENGTH, Y_LENGTH
      parameter (DFACC_RDONLY = 1, X_LENGTH = 15, Y_LENGTH = 10)
      integer*2 all_image(2, X_LENGTH, Y_LENGTH)
      integer*2 subset_image(2, 3, 7)

C     Open the file.
      file_id = hopen('Example2.hdf', DFACC_RDONLY, 0)

C     Initialize the GR interface.
      gr_id = mgstart(file_id)

C     Select the first data set.
      ri_id = mgselct(gr_id, 0)

C     Read the entire data set.
      start(1) = 0
      start(2) = 0
      edges(1) = X_LENGTH
      edges(2) = Y_LENGTH
      stride(1) = 1
      stride(2) = 1
      status = mgrdimg(ri_id, start, stride, edges, all_image)

C     Read a subset from the middle of the data set.
      start(1) = 1
      start(2) = 1
      edges(1) = 3
      edges(2) = 7
      status = mgrdimg(ri_id, start, stride, edges, subset_image)

C     Terminate access to the array.
      status = mgendac(ri_id)

C     Terminate access to the GR interface and the file.
      status = mgend(gr_id)

C     Close the file.
      file_id = hclose(file_id)

      end
```

EXAMPLE 5.

Sampling Image Data

These examples illustrate how samples of data set elements can be read by using the `stride` argument of the **GRreadimage** function. Here we read every fourth row and every other column.

C:

```
#include "hdf.h"
#include "mfgr.h"

#define X_LENGTH 10
#define Y_LENGTH 20

main( )
{
int32 gr_id, ri_id, file_id, start[2], edges[2], ncomp, il;
int32 stride[2], dims[2];
int16 all_image[Y_LENGTH][X_LENGTH][2], sample_image[5][5][2], status;
intn i, j;

/* Open the file. */
file_id = Hopen("Example5.hdf", DFACC_CREATE, 0);

/* Open the file. */
gr_id = GRstart(file_id);

/* Define the number of components and dimensions of the image. */
ncomp = 2;
dims[0] = X_LENGTH;
dims[1] = Y_LENGTH;
il = MFGR_INTERLACE_PIXEL;

/*  Create the array. */
ri_id = GRcreate(gr_id, "Image_array_5", ncomp, DFNT_INT16, il, dims);

/*  Compute and store the data values. */
for (j = 0; j < Y_LENGTH; j++) {
    for (i = 0; i < X_LENGTH; i++) {
        all_image[j][i][0] = i + j * 10;
        all_image[j][i][1] = i + j * 10;
    }
}

/* Define the start and edge parameters. */
start[0] = start[1] = 0;
edges[0] = X_LENGTH;
edges[1] = Y_LENGTH;

/* Write the buffered data in all_image to the array. */
status = GRwriteimage(ri_id, start, NULL, edges, (VOIDP)all_image);

/* Close the GR interface and the file, then re-open both and
select the first and only data set in the file. */
status = GRendaccess(ri_id);
status = GRend(gr_id);
status = Hclose(file_id);
file_id = Hopen("Example5.hdf", DFACC_RDONLY, 0);
gr_id = GRstart(file_id);
ri_id = GRselect(gr_id, 0);

/* Read the data into sample_image, skipping every fourth \
row and every other column. */
```

```
start[0] = start[1] = 0;
edges[0] = 5;
edges[1] = 5;
stride[0] = 2;
stride[1] = 4;
status = GRreadimage(ri_id, start, stride, edges, (VOIDP)sample_image);

/* Terminate access to the array */
status = GRendaccess(ri_id);

/* Terminate access to the GR interface the file */
status = GRend(gr_id);

/* Close the file */
status = Hclose(file_id);

}
```

FORTRAN:

```
     PROGRAM READ STRIDES

      integer*4 gr_id, ri_id, file_id, ncomp, il
      integer*4 start(2), edges(2), stride(2), dims(2)
      integer i, j, status
      integer mgstart, mgcreat, mgwrimg, mgendac
      integer mgselct, mgend, hopen, hclose

C     DFACC_CREATE and DFNT_INT16 are defined in hdf.h.
      integer*4 DFACC_CREATE, DFACC_RDONLY, DFNT_INT16
      integer*4 X_LENGTH, Y_LENGTH

      parameter (DFACC_CREATE = 4, DFACC_RDONLY = 1, DFNT_INT16 = 22,
     +           X_LENGTH = 10, Y_LENGTH = 20)
      integer*2 all_image(2, X_LENGTH, Y_LENGTH)

C     Create the file.
      file_id = hopen('Example5.hdf', DFACC_CREATE, 0)

C     Create the file.
      gr_id = mgstart(file_id)

C     Define the number of components and dimensions of the image.
      ncomp = 2
      dims(1) = X_LENGTH
      dims(2) = Y_LENGTH

C     Create the array.
      ri_id = mgcreat(gr_id, 'Image_array_5', ncomp, DFNT_INT16, il,
     +           dims)

C     Compute and store the data values.
      do 20 j = 1, Y_LENGTH
         do 10 i = 1, X_LENGTH
            all_image(1, i, j) = i + j * 10
            all_image(2, i, j) = i + j * 10
10       continue
20    continue

C     Set up the start and edge parameters to write the buffered
C     data to the entire array.
      start(1) = 0
      start(2) = 0
```

```
      edges(1) = X_LENGTH
      edges(2) = Y_LENGTH
      stride(1) = 1
      stride(2) = 1

C     Write the buffered data in all_image to the data set array.
      status = mgwrimg(ri_id, start, stride, edges, all_image)

C     Close the GR interface and the file, then re-open both and
C     select the first and only data set in the file.
      status = mgendac(ri_id)
      status = mgend(gr_id)
      status = hclose(file_id)
      file_id = hopen('Example5.hdf', DFACC_RDONLY, 0)
      gr_id = mgstart(file_id)
      ri_id = mgselct(gr_id, 0)

C     Read the data into sample_image, skipping every fourth row
C     and every other column.
      edges(1) = 5
      edges(2) = 5
      stride(1) = 4
      stride(2) = 2

C     Terminate access to the array.
      status = mgendac(ri_id)

C     Terminate access to the GR interface.
      status = mgend(gr_id)

C     Close the file.
      status = hclose(file_id)

      end
```

8.7 Obtaining Information About Files and General Raster Images

The routines covered in this section provide methods for obtaining information about the images in a file, for identifying images that meet certain criteria, and for obtaining information about the data sets themselves.

GRfileinfo gives the number of images and file attributes in a file, and **GRgetiminfo** provides information about individual images. To cycle through the images in a file, a calling program must use **GRfileinfo** to determine the number of images, followed by repeated calls to **GRgetiminfo** to view them. The parameters for these routines are described below. (See Table 8F on page 219.)

8.7.1 Obtaining Information about the Contents of a File: GRfileinfo

It is often useful to determine the number of images and global file attributes contained in a file before executing a series of read or write operations. The **GRfileinfo** routine is designed for this purpose. To determine the contents of a file, the calling program must contain the following:

C:

```
file_id = Hopen(filename, access_mode, number_of_desc);
gr_id = GRstart(file_id);
status = GRfileinfo(gr_id, &n_datasets, &n_file_attr);
status = GRend(gr_id);
status = Hclose(file_id);
```

FORTRAN:
```
file_id = hopen(filename, access_mode, number_of_desc)
gr_id = mgstart(filename, access_mode)
status = mgfinfo(gr_id, n_datasets, n_file_attr)
status = mgend(gr_id)
status = hclose(file_id)
```

The number of images in the file will be returned in the `n_datasets` argument, and `n_file_attr` will contain the total number of file attributes.

8.7.2 Obtaining Information About an Image: GRgetiminfo

It is impossible to allocate the proper amount of memory to buffer the image data when the number of components, dimension sizes and/or data type of the image are unknown. The **GRgetiminfo** routine provides necessary information about images for this purpose. To access information about an image, the calling program must contain the following:

C:
```
file_id = Hopen(filename, access_mode, number_of_desc);
gr_id = GRstart(file_id);
ri_id = GRselect(gr_id, gr_index);
status = GRgetiminfo(ri_id, name, &ncomp, &data_type, &il, dim_sizes,
         &n_attr);
status = GRend(gr_id);
status = GRendaccess(ri_id);
status = Hclose(file_id);
```

FORTRAN:
```
file_id = hopen(filename, access_mode, number_of_desc)
gr_id = mgstart(file_id)
ri_id = mgselct(gr_id, gr_index)
status = mggiinf(ri_id, name, ncomp, data_type, il, dim_sizes,
         n_attr)
status = mgend(gr_id)
status = mgendac(ri_id);
status = hclose(file_id)
```

GRgetiminfo takes a raster image identifier as input, and returns the name, number of components, data type, interlace mode, dimension size and number of attributes for the corresponding image in the `name`, `ncomp`, `nt`, `il`, `dim_sizes` and `n_attr` arguments respectively. The number of components of an image array element corresponds to the order of a vdata field, therefore this implementation of image components in the GR interface is flexible enough to accommodate any representation of pixel data. The calling program determines this representation; the GR interface recognizes only the raw byte configuration of the data. The attribute count will only reflect the number of attributes assigned to the image array; file attributes are not included.

TABLE 8F **GRfileinfo and GRgetiminfo Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
GRfileinfo (mgfinfo)	gr_id	int32	integer	General raster data set identifier.
	n_datasets	int32 *	integer	Number of data sets in the file.
	n_file_attr	int32 *	integer	Number of global attributes in the file.
GRgetiminfo (mggiinf)	ri_id	int32	integer	Image identifier.
	name	char *	character* (*)	Buffer for the name of the image.
	ncomp	int32 *	integer	Buffer for the number of components in the image.
	data_type	int32 *	integer	Buffer for the data type for the data in the image.
	il	int32 *	integer	Buffer for the interlace type of the data in the image.
	dim_sizes	int32 [2]	integer (2)	Buffer for the size of each dimension in the image.
	nattrs	int32 *	integer	Buffer for the number of attributes in the image.

EXAMPLE 6.

Retrieving Image Information

The following examples illustrate the use of the **GRgetiminfo** routine.

C:

```
#include "hdf.h"
#include "mfgr.h"

#define X_LENGTH 15
#define Y_LENGTH 10

main( )
{
int32 gr_id, ri_id, file_id, status, il;
int32 nt, dimsizes[2], ncomp, nattrs;
int32 start[2], edges[2];
int16 image_data[Y_LENGTH][X_LENGTH][2];
char name[MAX_GR_NAME];

/* Open the file. */
file_id = Hopen("Example2.hdf", DFACC_RDONLY, 0);

/* Initiate the GR interface. */
gr_id = GRstart(file_id);

/* Select the first (and in this case, only) image in the file. */
ri_id = GRselect(gr_id, 0);

/* Verify the characteristics of the image. */
status = GRgetiminfo(ri_id, name, &ncomp, &nt, &il, dimsizes, &nattrs);

/* Define the location, pattern, and size of the data to read from \
    the image. */
start[0] = start[1] = 0;
edges[0] = dimsizes[0];
edges[1] = dimsizes[1];

/* Read the array data set created in Example 2. */
status = GRreadimage(ri_id, start, NULL, edges, (VOIDP)image_data);

/* Terminate access to the image. */
status = GRendaccess(ri_id);

/* Terminate access to the GR interface. */
status = GRend(gr_id);

/* Close the file. */
status = Hclose(file_id);

}
```

FORTRAN:

```
     PROGRAM GET IMAGE INFO

      integer*4 gr_id, ri_id, file_id, il, ncomp
      integer*4 data_type, nattrs, status
      integer start(2), edge(2), stride(2)
      integer mgstart, mgselct, mggiinf
      integer mgrdimg, mgendac, mgend, hopen, hclose
      integer*4 DFACC_RDONLY, MAX_GR_NAME
```

```
      integer*4 X_LENGTH, Y_LENGTH
      parameter (DFACC_RDONLY = 1, MAX_GR_NAME = 256,
     +           X_LENGTH = 15, Y_LENGTH = 10)
      integer*2 image_data(2, X_LENGTH, Y_LENGTH)
      character name(MAX_GR_NAME)
      integer*4 dimsizes(2)

C     Open the file.
      file_id = hopen('Example2.hdf', DFACC_RDONLY, 0)

C     Initiate the GR interface.
      gr_id = mgstart(file_id)

C     Select the first (and in this case, only) image in the file.
      ri_id = mgselct(gr_id, 0)

C     Verify the characteristics of the image.
      status = mggiinf(ri_id, name, ncomp, data_type, il, dimsizes,
     +            nattrs)

C     Define the location, pattern, and size of the data to read
C     from the image.
      start(1) = 0
      start(2) = 0
      stride(1) = 1
      stride(2) = 1
      edge(1) = dimsizes(1)
      edge(2) = dimsizes(2)

C     Read the image.
      status = mgrdimg(ri_id, start, stride, edge, image_data)

C     Terminate access to the array data set.
      status = mgendac(ri_id)

C     Terminate access to the GR interface.
      status = mgend(gr_id)

C     Close the file.
      status = hclose(file_id)

      end
```

EXAMPLE 7.

Printing Image Names

The **GRgetiminfo** function can be called within a loop to retrieve the names of all images in an HDF file, as shown in the following examples.

C:

```
#include "hdf.h"

main( )
{
int32 gr_id, ri_id, file_id, n_datasets, n_file_attrs, gr_index;
int32 dim_sizes[2];
int32 ncomp, il, data_type, attributes, status;
char name[MAX_GR_NAME];

/* Open the file. */
```

```
file_id = Hopen("Example2.hdf", DFACC_RDONLY, 0);

/* Initiate the GR interface. */
gr_id = GRstart(file_id);

/* Determine the contents of the file. */
status = GRfileinfo(gr_id, &n_datasets, &n_file_attrs);

/* Access and print the name of each image in the file. */
for (gr_index = 0; gr_index < n_datasets; gr_index++) {
    ri_id = GRselect(gr_id, gr_index);
    status = GRgetiminfo(ri_id, name, &ncomp, &data_type, \
                &il, dim_sizes, &attributes);
    printf("name = %s\n", name);
    status = GRendaccess(ri_id);
}

/* Terminate access to the GR interface. */
status = GRend(gr_id);

/* Close the file. */
status = Hclose(file_id);

}
```

FORTRAN:

```
     PROGRAM PRINT NAMES

      integer*4 gr_id, ri_id, file_id
      integer*4 n_datasets, n_file_attrs, index, status
      integer*4 attributes
      integer mgstart, mgfinfo, mgselct, mggiinf
      integer mgendac, mgend, hopen, hclose

      integer*4 DFACC_RDONLY, MAX_GR_NAME
      parameter (DFACC_RDONLY = 1, MAX_GR_NAME = 256)

      integer*4 dim_sizes(2)
      character name *(MAX_GR_NAME)

C     Open the file.
      file_id = hopen('Example2.hdf', DFACC_RDONLY, 0)

C     Initiate the GR interface.
      gr_id = mgstart(file_id)

C     Determine the contents of the file.
      status = mgfinfo(gr_id, n_datasets, n_file_attrs)

C     Access and print the names of every data set in the file.
      do 10 gr_index = 0, n_datasets - 1
         ri_id = mgselct(gr_id, index)
         status = mggiinf(ri_id, name, ncomp, data_type, il,
     +                    dim_sizes, attributes)
         print *, "name = ", name
         status = mgendac(ri_id)
10    continue

C     Terminate access to the GR interface.
      status = mgend(gr_id)

C     Close the file.
```

```
status = hclose(file_id)

end
```

8.7.3 Getting the Index of an Image: GRreftoindex and GRnametoindex

In addition to an index and name, images are also assigned a tag and reference number. **GRrefto-index** determines the index of the image from its reference number.

Selecting a data set from a given reference number involves the following steps:

1. Open the file.
2. Convert the reference number for the image into an index number.
3. Select the image by obtaining its identifier from its index number.

The calling program must contain the following:

C:
```
gr_id = GRstart(file_id);
gs_index = GRreftoindex(gr_id, ref);
ri_id = GRselect(gr_id, gs_index);
```

FORTRAN:
```
gr_id = mgstart(file_id)
gs_index = mgr2idx(gr_id, ref)
ri_id = mgselct(gr_id, gs_index)
```

GRreftoindex returns the index specified by the `gs_index` parameter for the image in the file with the reference number `ref`. The index `gs_index` can then be passed to **GRselect** to obtain an identifier for the image.

Images can also be selected using the image name as the search criteria. The calling sequence described above is the same for this purpose, except **GRnametoindex** is called instead of **GRref-toindex** and the image name is passed as the second parameter to **GRnametoindex**. As with **GRreftoindex**, **GRnametoindex** returns the index number of the target image.

TABLE 8G **GRreftoindex and GRnametoindex Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
GRreftoindex (mgr2idx)	gr_id	int32	integer	General raster data set identifier.
	ref	uint16	integer	Reference number of the target image.
GRnametoindex (mgn2ndx)	gr_id	int32	integer	General raster data set identifier.
	name	char *	character (*)	Name of the target image.

EXAMPLE 8. **Searching for an Image Index**

If the index of a image isn't known, the **GRnametoindex** function can be used to retrieve the index based on a given data set name. In these examples, an invalid image name is given which results in **GRnametoindex** returning a value of `-1`. A valid image name is then provided and the returned index is used to read the contents of the corresponding image.

C:

```
#include "hdf.h"

#define X_LENGTH 10
```

```
#define Y_LENGTH 20

main( )
{

int32 gr_id, gr_index, ri_id, file_id, status;
int32 start[2], edges[2];
int16 image_data[Y_LENGTH][X_LENGTH][2];

/* Open the file. */
file_id = Hopen("Example5.hdf", DFACC_RDONLY, 0);

/* Initiate the GR interface. */
gr_id = GRstart(file_id);

/* Search for the index of a non-existent image. */
gr_index = GRnametoindex(gr_id, "Invalid_Data_Set_Name");

/* Error condition: gr_index contains the value -1. */

/* Search for the index of the image named "Image_array_5". */
gr_index = GRnametoindex(gr_id, "Image_array_5");

/* Select the image corresponding to the returned index. */
ri_id = GRselect(gr_id, gr_index);

/* Read the data set data into the array_data array. */
start[0] = start[1] = 0;
edges[0] = X_LENGTH;
edges[1] = Y_LENGTH;
status = GRreadimage(ri_id, start, NULL, edges, (VOIDP)image_data);

/* Terminate access to the array */
status = GRendaccess(ri_id);

/* Terminate access to the GR interface. */
status = GRend(gr_id);

/* Close the file */
 status = Hclose(file_id);

}
```

FORTRAN:

```
     PROGRAM SEARCH INDEX

      integer*4 gr_id, gr_index, ri_id, file_id, status
      integer*4 start(2), edges(2), stride(2)
      integer mgstart, mgn2ndx, mgrdimg, mgendac, mgselct
      integer mgend, hopen, hclose

C     DFACC_RDONLY is defined in hdf.h.
      integer*4 DFACC_RDONLY
      integer*4 X_LENGTH, Y_LENGTH
      parameter (DFACC_RDONLY = 1, X_LENGTH = 10, Y_LENGTH = 20)
      integer*2 image_data(2, X_LENGTH, Y_LENGTH)

C     Open the file.
      file_id = hopen('Example5.hdf', DFACC_RDONLY, 0)

C     Initialize the GR interface.
      gr_id = mgstart(file_id)
```

```
C     Search for the index of a non-existent image.
      gr_index = mgn2ndx(gr_id, 'Invalid_Data_Set_Name')

C     Error condition: gr_index contains the value -1.
C     Search for the index of the image named 'Image_array_5'.
      gr_index = mgn2ndx(gr_id, 'Image_array_5')

C     Select the image corresponding to the returned index.
      ri_id = mgselct(gr_id, gr_index)

C     Read the image data into the image_data array.
      start(1) = 0
      start(2) = 0
      edges(1) = X_LENGTH
      edges(2) = Y_LENGTH
      stride(1) = 1
      stride(2) = 1
      status = mgrdimg(ri_id, start, stride, edges, image_data)

C     Terminate access to the array.
      status = mgendac(ri_id)

C     Terminate access to the GR interface and the file.
      status = mgend(gr_id)

C     Close the file.
      status = hclose(file_id)

      end
```

8.7.4 Compressing General Raster Images

Images can be compressed by calling the **GRsetcompress** routine. **GRsetcompress** compresses the image data at the time it is called and supports all standard HDF compression algorithms. The syntax of the **GRsetcompress** routine is as follows:

C: `status = GRsetcompress(ri_id, comp_type, c_info);`

As with **SDsetcompress**, the `comp_type` argument is the compression type definition and all compression method definitions supported in **SDsetcompress** are also supported in **GRsetcompress**. The `c_info` argument is a structure of type `comp_info` - this structure is defined in the "hcomp.h" header file and in Chapter 3 titled *Scientific Data Sets (SD API)*. It contains the algorithm-specific information necessary for **GRsetcompress** to perform the specified data compression.

TABLE 8H **GRsetcompress Parameter List**

Routine Name	Parameter	Data Type C	Description
GRsetcompress	ri_id	int32	Image identifier.
	comp_type	int32	Compression method.
	c_info	comp_info *	Pointer to compresion infomation structure.

8.7.5 External File Operations Using the GR API

An ***external image array*** is one that is stored in a file that is not the file containing the metadata for the image. The concept of externally stored data is described in Chapter 3, titled *Scientific Data Sets (SD API)*. The GR interface supports the same functionality as the SD interface.

8.7.5.1 Creating a General Raster Image in an External File

Creating an image using external data involves the same general steps as with the SD interface:

1. Create the image array.
2. Specify that an external data file is to be used.
3. Write data to the image array.
4. Terminate access to the image.

To create a data set containing an external file, the calling program must make the following calls.

C:
```
ri_id = GRcreate(gr_id, name, ncomp, nt, il, dim_sizes);
status = GRsetexternalfile(ri_id, filename, offset);
status = GRwriteimage(ri_id, start, stride, edge, image_data);
status = GRendaccess(ri_id);
```

FORTRAN:
```
ri_id = mgcreat(gr_id, name, ncomp, nt, il, dim_sizes)
status = mgsxfil(ri_id, filename, offset)
status = mgwrimg(ri_id, start, stride, edge, image_data)
status = mgendac(ri_id)
```

GRsetexternalfile marks the image identified by `ri_id` as one whose data is to be written to an external file. The parameter `filename` is the name of the external data file, and `offset` is the number of bytes from the beginning of the external file to the location where the first byte of data will be written.

GRsetexternalfile can only be called once per data set. If a file with the same name as `filename` exists in the current directory, HDF will use it as the external file. If the file does not exist, HDF will create one. Once the external filename is specified, it is impossible to change it without breaking the association between the general raster data set and its image data.

Use caution when writing to existing files because the **GRwriteimage** library begins its write at the specified offset without checking whether data is being overwritten. When different data sets have arrays occupying the same external file, the calling program is responsible for avoiding any overlap between them.

For more information on the parameters used in **GRsetexternalfile** refer to the following table.

TABLE 8I **GRsetexternalfile Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
GRsetexternalfile (mgsxfil)	ri_id	int32	integer	Image identifier.
	filename	char *	character* (*)	Name of the external data set.
	offset	int32	integer	Offset in bytes from the beginning of the external file to the image data

8.7.5.2 Moving General Raster Images to an External File

Images can be moved from a primary file to an external file. To do so requires the following steps:

1. Select the image.
2. Specify the external data file.
3. Terminate access to the image.

The calling program must make the following calls:

C:
```
ri_id = GRselect(gr_id, gr_index);
status = GRsetexternalfile(ri_id, filename, offset);
status = GRendaccess(ri_id);
```

When **GRsetexternalfile** is used in conjunction with **GRselect**, it will immediately write the existing data to the external file and any data in the external file that occupies the space reserved for the external array will be overwritten as a result of this operation. A data set can only be moved to an external file once.

During the operation, the data is written to the external file as a contiguous stream regardless of how it is stored in the primary file. Because data is moved "as is", any unwritten locations in the data set are preserved in the external file. Subsequent read and write operations performed on the data set will access the external file.

8.7.5.3 Setting the I/O Mode for External General Raster Image Access

In situations where it is necessary to perform parallel I/O access to external image objects - for instance, if the external image data resides on a RAID array and is to be accessed by a multiprocessor system - the **GRsetaccesstype** routine can be used to set the access mode to either parallel or serial I/O. The syntax of the **GRsetaccesstype** routine is as follows:

C:
```
status = GRsetaccesstype(ri_id, access_mode);
```

FORTRAN:
```
status = mgsactp(ri_id, access_mode)
```

The `access_mode` argument can be set to either `DFACC_SERIAL` for serial mode or `DFACC_PARALLEL` for parallel mode. **GRsetaccesstype** returns a status code of `0` for successful completion or `1` if an error occurred.

TABLE 8J **GRsetaccesstype Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
GRsetaccesstype (mgsactp)	ri_id	int32	integer	Image identifier.
	access_mode	char *	character* (*)	Access mode.

8.8 General Raster Image Attributes

The concepts of user-defined and predefined attributes are explained in Chapter 3, titled *Scientific Data Sets (SD API)*. The GR implementation of attributes is similar to the SD implementation. Attributes are not written out to a file until access to the object the attribute is attached to is terminated.

8.8.1 Predefined GR Attributes

The GR interface library has only one predefined attribute: `FILL_ATTR`. This attribute defines a fill pixel, which is analogous to a fill value in the SD interface. It represents the default value that is written to each element of an image array not explicitly written to by the calling program; when only a portion of the entire image array is filled with data. This value must of the same data type as the rest of the initialized image data. The routine used to set the fill value is **GRsetattr**, explained in the next section.

8.8.2 Setting User-Defined Attributes: GRsetattr

Setting an attribute involves the following steps:

1. Open the file.
2. Initialize the GR interface.
3. Set the attribute.
4. Terminate access by disposing of any existing identifiers.
5. Terminate access to the GR interface.
6. Close the file.

To assign an attribute to a file, the calling program must contain the following sequence calls:

C:
```
gr_id = GRstart(filename, access_mode);
status = GRsetattr(gr_id|ri_id, attr_name, data_type, count,
        attr_value);
status = GRend(gr_id);
```

FORTRAN:
```
gr_id = mgstart(filename, access_mode)
status = mgsnatt(gr_id|ri_id, attr_name, data_type, count,
        attr_value)
status = mgend(gr_id)
```

In **GRsetattr** the first argument can either be the id of a file, palette or raster image. The attribute will be assigned to the object referred to by this identifier. The argument `attr_name` contains the name of the attribute and can be no more than `MAX_GR_NAME` characters in length. Passing the name of an existing attribute will overwrite the value portion of that attribute.

The arguments, `data_type`, `count` and `attr_value` describe the right side of the `label=value` equation. The `attr_value` argument contains one or more values of the same data type. The `data_type` argument describes the data type for all values in the attribute and `count` contains the total number of values in the attribute.

As with other HDF routines that accommodate numeric and character data, the **GRsetattr** routine has two Fortran-77 versions: **mgsnatt** and **mgscatt**. The **mgsnatt** routine writes numeric attribute data and **mgscatt** writes character attribute data.

The parameters for **GRsetattr** are further described below. (See Table 8K on page 231.) Note that, because there are two Fortran-77 versions of **GRsetattr**, there are correspondingly two entries in the "Data Type" field of the *values* parameter.

EXAMPLE 9. **Setting File and Image Attribute Values**

The following code segment calls **GRsetattr** to assign file and image attributes to a general raster data set.

C:

```
#include "hdf.h"

main( )
```

```
{
int32 gr_id, ri_id, file_id, dim_index, status;
int32 num_values[2];

/* Open the file. */
file_id = Hopen("Example5.hdf", DFACC_RDWR, 0);

/* Initialize the GR interface. */
gr_id = GRstart(file_id);

/* Set an attribute that describes the file contents. */
status = GRsetattr(gr_id, "File contents", DFNT_CHAR8, 16, \
                (VOIDP)"RGB image data");

/* Get the identifier for the first data set. */
ri_id = GRselect(gr_id, 0);

/* Set an attribute the specifies a valid range of values. */
num_values[0] = 2;
num_values[1] = 10;
status = GRsetattr(ri_id, "Value range", DFNT_INT32, 2, \
                (VOIDP)num_values);

/* Terminate access to the image. */
status = GRendaccess(ri_id);

/* Terminate access to the GR interface. */
status = GRend(gr_id);

/* Close the file. */
status = Hclose(file_id);

}
```

FORTRAN:

```
     PROGRAM SET IMAGE ATTRIBS

      integer*4 gr_id, ri_id, file_id, status
      integer num_values(2)
      integer mgstart, mgsnatt, mgselct
      integer mgendac, mgend, hopen, hclose
      integer DFACC_RDWR, DFNT_CHAR8, DFNT_INT32
      parameter (DFACC_RDWR = 3, DFNT_CHAR8 = 4, DFNT_INT32 = 24)

C     Open the file.
      file_id = hopen('Example5.hdf', DFACC_RDWR, 0)

C     Initialize the GR interface.
      gr_id = mgstart(file_id)

C     Set an attribute that describes the file contents.
      status = mgsnatt(gr_id, 'File contents', DFNT_CHAR8, 16,
     +                 'RGB image data')

C     Get the identifier for the first data set.
      ri_id = mgselct(gr_id, 0)

C     Set an attribute the specifies a valid range of values.
      num_values(1) = 2
      num_values(2) = 10
      status = mgsnatt(ri_id, 'Value range', DFNT_INT32, 2, num_values)
```

```
C     Terminate access to the image.
      status = mgendac(ri_id)

C     Terminate access to the GR interface.
      status = mgend(gr_id)

C     Close the file.
      status = hclose(file_id)

      end
```

8.8.3 Querying User-Defined Attributes: GRfindattr and GRattrinfo

Each attribute associated with an object has a unique ***attribute index***, a value ranging from 0 to the total number of attributes in the object. Given a file, array or dimension id and an attribute name, **GRfindattr** will return a valid attribute index if the attribute exists. The attribute index can then be used to retrieve information about an attribute or its values. Given a file or array and a valid attribute index **GRattrinfo** returns the name, data type and count for the attribute if it exists.

The syntax for **GRfindattr** and **GRattrinfo** is as follows:

C:
```
status = GRfindattr(ri_id|gr_id, attr_name);
status = GRattrinfo(ri_id|gr_id, attr_index, attr_name,
        &data_type, &count);
```

FORTRAN:
```
status = mgfndat(ri_id|gr_id, attr_name)
status = mgatinf(ri_id|gr_id, attr_index, attr_name,
        data_type, count);
```

The parameters for **GRfindattr** and **GRattrinfo** are further described below. (See Table 8K.)

8.8.4 Reading User-Defined Attributes: GRgetattr

GRgetattr reads an attribute's values. The syntax for **GRgetattr** is as follows:

C:
```
status = GRgetattr(ri_id|gr_id, attr_index, data);
```

FORTRAN:
```
status = mggnatt(ri_id|gr_id, attr_index, data)
```

GRgetattr takes a file, array, or dimension identifier and an attribute index specified in the `attr_index` parameter as input parameters and returns the attribute's values in the buffer `data`.

It's assumed that the buffer `data`, allocated to hold the attribute values, is large enough to hold the data - if not, the data read will be truncated to the size of the buffer. The size of the buffer should be at least `count*DFKNTsize(number_type)` bytes long. If an attribute contains multiple values, **GRgetattr** will return all of them. It is not possible to read a subset of values.

As with **GRsetattr**, **GRgetattr** has two Fortran-77 versions: **mggnatt** and **mggcatt**. The **mggnatt** routine reads numeric attribute data and **mggcatt** reads character attribute data.

The parameters for **GRgetattr** are further described in Table 8K. Note that, because there are two Fortran-77 versions of **GRgetattr**, there are correspondingly two entries in the "Data Type" field of the *values* parameter.

TABLE 8K

GRsetattr, GRfindattr, GRattrinfo and GRgetattr Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
GRsetattr (mgsnatt/ mgscatt)	*_id	int32	integer	GR data set or image identifier.
	attr_name	char *	character* (*)	Name assigned to the attribute.
	data_type	int32	integer	Specifies the data type.
	count	int32	integer	Number of values in the attribute.
	values	VOIDP	<valid numeric data type>	Buffer of the data to be written.
GRfindattr (mgfadat)	*_id	int32	integer	GR data set or image identifier.
	attr_name	char *	character* (*)	Name assigned to the attribute.
GRattrinfo (mgatinf)	*_id	int32	integer	GR data set or image identifier.
	attr_index	int32	integer	Index for the attribute to be read.
	attr_name	char *	character* (*)	Buffer for the name of the dimension attribute.
	data_type	int32 *	integer	Buffer for the data type of the values in the attribute.
	count	int32 *	integer	Buffer for the total number of values in the attribute.
GRgetattr (mggnatt/ mggcatt)	*_id	int32	integer	GR data set or image identifier.
	attr_index	int32	integer	Index for the attribute to be read.
	values	VOIDP	<valid numeric data type>	Buffer for the attribute values.

EXAMPLE 10.

Retrieving Image Attribute Information

The image attribute information stored in the "Example4.hdf" HDF file in Example 4 are read from the file in these examples.

C:

```
#include "hdf.h"

main( )
{

int32 gr_id, ri_id, file_id, status;
int32 attr_index, data_type, count;
char attr_name[MAX_GR_NAME];
int8 *buffer;

/* Open the file. */
file_id = Hopen("Example5.hdf", DFACC_RDONLY, 0);

/* Initialize the GR interface. */
gr_id = GRstart(file_id);

/* Find the file attribute named "File contents". */
attr_index = GRfindattr(gr_id, "File contents");

/* Get information about the file attribute. */
status = GRattrinfo(gr_id, attr_index, attr_name, &data_type, &count);

/* Allocate a buffer to hold the attribute data. */
buffer = HDmalloc(count * DFKNTsize(data_type));

/* Read the attribute data. */
status = GRgetattr(gr_id, attr_index, buffer);
```

```
/* Free the buffer memory. */
HDfree(buffer);

/* Get the identifier for the first image. */
ri_id = GRselect(gr_id, 0);

/* Find the data set attribute named "Value range". */
attr_index = GRfindattr(ri_id, "Value range");

/* Get information about the image attribute. */
status = GRattrinfo(ri_id, attr_index, attr_name, &data_type, &count);

/* Allocate a buffer to hold the attribute data. */
buffer = HDmalloc(count * DFKNTsize(data_type));

/* Read the attribute data. */
status = GRgetattr(ri_id, attr_index, buffer);

/* Terminate access to the array */
status = GRendaccess(ri_id);

/* Terminate access to the GR interface. */
status = GRend(gr_id);

/* Close the file */
status = Hclose(file_id);

}
```

FORTRAN:

```
     PROGRAM IMAGE ATTRIB INFO

      integer*4 gr_id, ri_id, file_id
      integer attr_index, data_type, count, status
      character attr_name *13
      character buffer *20

      integer mgstart, mgfndat, mgatinf, mggnatt, mgselct
      integer mgendac, mgend, hopen, hclose

C     DFACC_RDONLY is defined in hdf.h.
      integer DFACC_RDONLY
      parameter (DFACC_RDONLY = 1)

C     Open the file.
      file_id = hopen('Example5.hdf', DFACC_RDONLY, 0)

C     Open the file.
      gr_id = mgstart(file_id)

C     Find the file attribute named 'File contents'.
      attr_index = mgfndat(gr_id, 'File contents')

C     Get information about the file attribute.
      status = mgatinf(gr_id, attr_index, attr_name, data_type, count)

C     Read the attribute data.
      status = mggnatt(gr_id, attr_index, buffer)

C     Get the identifier for the first image.
      ri_id = mgselct(gr_id, 0)
```

```
C     Find the data set attribute named "Value range".
      attr_index = mgfndat(ri_id, 'Value range')

C     Get information about the image attribute.
      status = mgatinf(ri_id, attr_index, attr_name, data_type, count)

C     Read the attribute data.
      status = mggnatt(ri_id, attr_index, buffer)

C     Terminate access to the array.
      status = mgendac(ri_id)

C     Terminate access to the GR interface.
      status = mgend(gr_id)

C     Close the file.
      status = hclose(file_id)
      end
```

8.9 Reading and Writing Palette Data Using the GR API

The GR API library includes routines that read, write and access information about palette data attached to general raster images. Although this functionality is also provided by the HDF Palette API library it is not a recommended practice to use the Palette API to access and manipulate palette objects created by GR API routines.

The routines are named **GRgetlutid**, **GRgetlutinfo**, **GRluttoref**, **GRwritelut** and **GRreadlut**. Note that the routine names use the term ***LUT*** to refer to palettes, which stands for color ***lookup tables***.

8.9.1 Obtaining a Palette ID: GRgetlutid

The **GRgetlutid** function takes two arguments, the raster image identifier of the image that has the target palette attached to it and the index of the target palette, and returns the palette identifier corresponding to the specified image as its return value.

The syntax of the **GRgetlutid** function is:

C: `pal_id = GRgetlutid(ri_id, lut_index);`

FORTRAN: `pal_id = mggltid(ri_id, lut_index)`

8.9.2 Obtaining Palette Information: GRgetlutinfo

The **GRgetlutinfo** function takes one input argument, the identifier of the palette, and returns the number of components of the palette, the type of data contained in the palette, the interlace mode of the stored palette data and the number of entries in the palette in the `ncomp`, `data_type`, `interlace_mode` and `num_entries` arguments respectively. It returns a status code of `0` on successful completion and `-1` if an error occurred.

The syntax of the **GRgetlutinfo** function is:

C:
```
status = GRgetlutinfo(pal_id, &ncomp, &data_type, &interlace_mode,
         num_entries);
```

FORTRAN: `status = mgglinf(pal_id, ncomp, data_type, interlace_mode, num_entries)`

8.9.3 Retrieving the Reference Number of the Specified Palette: GRluttoref

The **GRluttoref** routine takes one argument, a palette identifier, and returns the reference number of the palette. **GRluttoref** is commonly used to annotate the palette or including the palette within a vgroup

There is currently no Fortran-77 version of the **GRluttoref**.

TABLE 8L **GRluttoref Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
GRluttoref	pal_id	int32	None	Image identifier.

8.9.4 Writing Palette Data: GRwritelut

The **GRwritelut** function takes six input arguments, the identifier of the palette, the number of components of the palette, the type of data contained in the palette, the interlace mode of the stored palette data, the number of entries in the palette and the buffered palette data to be written in the `pal_id`, `ncomp`, `data_type`, `interlace_mode`, `num_entries` and `pal_data` arguments respectively. It writes the palette data into the palette that is attached to the image specified by the given image identifier and returns a status code of `0` on successful completion and `-1` if an error occurred.

The syntax of the **GRwritelut** function is:

C: `status = GRwritelut(pal_id, ncomp, data_type, interlace_mode, num_entries, pal_data);`

FORTRAN: `status = mgwrlut(pal_id, ncomp, data_type, interlace_mode, num_entries, pal_data)`

There are two Fortran-77 versions of **GRwritelut**: **mgwrlut** and **mgwclut**. The **mgwrlut** routine writes buffered numeric palette data and **mgwclut** writes buffered character palette data.

8.9.5 Reading Palette Data: GRreadlut

The **GRreadlut** function takes two arguments, a palette identifier and a buffer large enough to store the read palette data. It reads the data into the buffer pointed to by the `pal_data` argument and returns a status code of `0` on successful completion and `-1` if an error occurred. The palette data is read according to the interlacing mode set by the last call to **GRreqlutil**.

The syntax of the **GRreadlut** function is:

C: `status = GRreadlut(pal_id, pal_data);`

FORTRAN: `status = mgrdlut(pal_id, pal_data)`

There are two Fortran-77 versions of **GRreadlut**: **mgrdlut** and **mgrclut**. The **mgrdlut** routine reads numeric palette data and **mgrclut** reads character palette data.

TABLE 8M

GRgetlutid, GRgetlutinfo, GRwritelut and GRreadlut Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
GRgetlutinfo (mgglinf)	pal_id	int32	integer	Palette identifier.
	ncomp	int32*	integer (*)	Buffer for the number of components in each palette element.
	data_type	int32*	integer (*)	Buffer for the data type of the palette data.
	interlace	int32*	integer (*)	Buffer for the interlace type of the palette data.
	num_entries	int32*	integer (*)	Buffer for the size of the palette.
GRwritelut (mgwrlut/ mgwclut)	pal_id	int32	integer	Palette identifier.
	ncomp	int32	integer	Number of components in each palette element.
	data_type	int32	integer	Type of the palette data.
	interlace	int32	integer	Interlace type of the palette data.
	num_entries	int32	integer	Number of entries in the palette.
	pal_data	VOIDP	<valid numeric data type>	Palette data to be written.
GRgetlutid (mggltid)	ri_id	int32	integer	General raster image identifier.
	index	int32	integer	Image index.
GRreadlut (mgrdlut/ mgrclut)	pal_id	int32	integer	Palette identifier.
	pal_data	VOIDP	<valid numeric data type>	Buffer for the palette data to be read.

EXAMPLE 11.

Reading and Writing a Palette and Obtaining Palette Information

In the following code examples, **GRgetlutid, GRgetlutinfo, GRwritelut and GRreadlut** are used to write a palette to an HDF file named "Example12.hdf", then read it into a buffer and retrieve information about it.

C:

```
#include "hdf.h"
#include "mfgr.h"

#define X_LENGTH 20
#define Y_LENGTH 20

main( )
{
int32 gr_id, ri_id, file_id, pal_id, status, num_entries;
int32 data_type, ncomp, num_comp, interlace_mode;
int32 r_num_entries, r_data_type, r_ncomp, r_interlace_mode;
uint8 palette_data[256*3];
uint8 r_palette_data[256*3];
intn i, j;
int32 dims[2];

/* Create and open the file. */
file_id = Hopen("Example11.hdf", DFACC_CREATE, 0);

/* Initiate the GR interface. */
gr_id = GRstart(file_id);
```

```
/* Define the number of components and dimensions of the image. */
ncomp = 1;
dims[0] = X_LENGTH;
dims[1] = Y_LENGTH;
interlace_mode = MFGR_INTERLACE_PIXEL;

/* Create the image. */
ri_id = GRcreate(gr_id, "Image_12", ncomp, DFNT_UINT8,
                interlace_mode, dims);

/* Initialize the palette to grayscale. */
for (i = 0; i < 256; i++) {
    palette_data[i * 3] = i;
    palette_data[i * 3 + 1] = i;
    palette_data[i * 3 + 2] = i;
}

/* Set palette characteristics. */
data_type = DFNT_UINT8;
num_entries = 256;
num_comp = 3;

/* Get the id for the palette. */
pal_id = GRgetlutid(ri_id, 0);

/* Write the palette to file. */
status = GRwritelut(pal_id, num_comp, data_type,
                interlace_mode, num_entries,
                (VOIDP)palette_data);

/* Read the palette information. */
status = GRgetlutinfo(pal_id, &r_ncomp, &r_data_type,
                &r_interlace_mode, &r_num_entries);

/* Read the palette data. */
status = GRreadlut(pal_id, (VOIDP)r_palette_data);

/* Terminate access to the image. */
status = GRendaccess(ri_id);

/* Terminate access to the GR interface. */
status = GRend(gr_id);

/* Close the file. */
status = Hclose(file_id);

}
```

FORTRAN:

```
     PROGRAM READ WRITE PALETTE

      integer*4 gr_id, ri_id, file_id, pal_id, status
      integer*4 num_entries, data_type, ncomp, interlace_mode
      integer*4 r_interlace_mode, r_num_entries, r_data_type
      integer*4 r_ncomp, ncomp, num_comp, dims(2)
      character palette_data(256*3)
      character r_palette_data(256*3)
      integer i
      integer*4 MFGR_INTERLACE_PIXEL, DFNT_CHAR8, X_LENGTH
      integer*4 Y_LENGTH
      parameter(MFGR_INTERLACE_PIXEL = 0, DFNT_CHAR8 = 4,
```

```
     +          X_LENGTH = 20, Y_LENGTH = 20)
      integer hopen, hclose, mgstart, mgcreat, mgwrlut
      integer mggltid, mgglinf, mgrdlut, mgendac, mgend

C     Create and open the file.
      file_id = hopen('Example11.hdf', DFACC_CREATE, 0)

C     Initiate the GR interface.
      gr_id = mgstart(file_id)

C     Define the Number of components and dimensions of the image
C     and palette to be created.
      ncomp = 1
      num_entries = 256
      num_comp = 3
      dims(1) = X_LENGTH
      dims(2) = Y_LENGTH
      interlace_mode = MFGR_INTERLACE_PIXEL
      data_type = DFNT_CHAR8

C     Create the image.
      ri_id = mgcreat(gr_id, 'Image_12', ncomp, data_type,
     +                interlace_mode, dims)

C     Initialize the palette to greyscale.
      do 10, i = 1, 256
        palette_data((i - 1) * 3 + 1) = char(i - 1)
        palette_data((i - 1) * 3 + 2) = char(i - 1)
        palette_data((i - 1) * 3 + 3) = char(i - 1)
10    continue

C     Get the identifier for the palette.
      pal_id = mggltid(ri_id, 0)

C     Write the palette to the HDF file.
      status = mgwrlut(pal_id, num_comp, data_type,
     +                 interlace_mode, num_entries,
     +                 palette_data)

C     Read the palette information.
      status = mgglinf(pal_id, ncomp, data_type,
     +                 interlace_mode, num_entries)

C     Read the palette data.
      status = mgrdlut(pal_id, r_ncomp, r_data_type,
     +                 r_interlace_mode, r_num_entries,
     +                 r_palette_data)

C     Terminate access to the image.
      status = mgendac(ri_id)

C     Terminate access to the GR interface.
      status = mgend(gr_id)

C     Close the file.
      status = hclose(file_id)

      end
```

8.10 GR Object Identifier Conversion Functions

There are situations where it is necessary to convert, or map, one type of object identifier to another. One instance is when an image is to be attached to a vgroup, another is when an image's index must be obtained so that it can be passed to the **GRselect** routine to access the image, but all that is available is the image's reference number. The former instance requires a means of mapping a raster image identifier to a reference number, which in turn will be stored in a vgroup object. The latter instance requires a means of mapping an image's reference number to an index. The **GRidtoref** and **GRreftoindex** functions provide this functionality.

8.10.1 Mapping an Image Identifier to a Reference Number: GRidtoref

The **GRidtoref** routine takes one argument, an image identifier, and maps it to a valid reference number.

8.10.2 Mapping a Reference Number to an Index: GRreftoindex

The **GRreftoindex** routine takes two arguments, a file identifier and the reference number of an image, and maps them to an image index.

TABLE 8N **GRidtoref and GRreftoindex Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
GRidtoref (mgid2rf)	ri_id	int32	integer	Image identifier.
GRreftoindex (mgr2idx)	gr_id	int32	integer	General raster data set identifier.
	ref	uint16	integer	Reference number to be mapped.

Chapter 9

Palettes (DFP API)

9.1 Chapter Overview

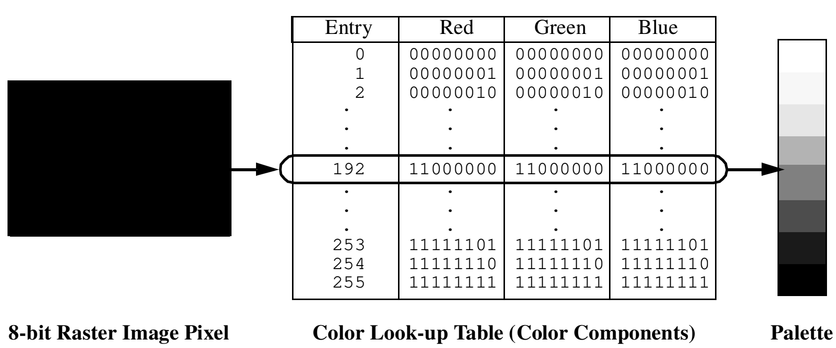
This chapter describes the routines available for storing and retrieving 8-bit palettes. An 8-bit palette is a look-up table with 256 entries, one entry for each of the 256 possible pixel values the system hardware associates with a particular color. This chapter introduces and describes the HDF palette data model and the DFP interface.

9.2 The Palette Data Model

A ***palette*** is the means by which color is applied to an image and is also referred to as a ***color lookup table***. It is a table in which every row contains the numerical representation of a particular color. Palettes can be many different sizes, but HDF only supports palettes with 256 colors, corresponding to the 256 different possible pixel values (0 to 255) in 8-bit raster images.

For each of the 256 colors in a palette, there are three 8-bit numbers describing its appearance. (See Figure 9a.) Each 8-bit color component represents the amount of red (or "R"), green (or "G"), or blue (or "B") used to create a particular color. In HDF, 8-bit palettes are assumed to be organized as follows; each entry consists of three bytes: one each for R, G, and B value. The first group of three bytes represent the R, G, and B values of the first color in the palette; the next three the R, G, and B values of the second color; and so forth. Therefore, the 256 possible different pixel values in an image serve as an index for the 256 color entries stored in the palette.

FIGURE 9a **Color Mapping Using a Palette**



In the HDF library, there are four interfaces that support the reading and writing of palette data; the raster image interfaces, covered in Chapter 6, titled *8-Bit Raster Images (DFR8 API)*, Chapter 7, titled *24-Bit Raster Images (DF24 API)*, Chapter 8, titled *General Raster Images (GR API)* and the DFP palette interface covered in this chapter. The raster image interfaces store palettes with raster images and the palette interface reads and writes palettes outside of raster image sets. Palettes stored using the palette interface are stored as isolated data objects. In other words they are not included as members of any set, although they can be grouped with other objects using the Vgroup interface. For more information on the Vgroup interface, refer to Chapter 5, titled *Vgroups (V API)*.

9.3 The Palette API

The DFP interface consists of eight routines. The routines **DFPaddpal** and **DFPgetpal** are the primary routines for palette I/O and are used for most reading and writing operations.

9.3.1 Palette Library Routines

All C functions in the palette interface are prefaced by "DFP" and the equivalent Fortran-77 functions are prefaced by "dp". These routines are divided into the following categories:

- ***Write routines*** store palettes in new files or append them to existing files.
- ***Read routines*** sequentially or randomly locate palettes to be read from a named file.

The DFP function calls are more explicitly defined in the following table and in the *HDF Reference Guide*.

TABLE 9A **DFP Library Routines**

Category	Routine Names		Description
	C	Fortran-77	
Write	**DFPaddpal**	**dpapal**	Appends a palette to a file.
	DFPputpal	**dpppal**	Writes a palette to a file.
	DFPwriteref	**dpwref**	Sets the reference number for writing the next palette.
Read	**DFPgetpal**	**dpgpal**	Retrieves the next palette in a file.
	DFPreadref	**dprref**	Sets reference number for retrieving the next palette.
	DFPlastref	**dplref**	Returns the value of the last reference number read or written.
	DFPnpals	**dpnpals**	Returns the number of palettes in a file.
	DFPrestart	**dprest**	Specifies that the next read call will get the first palette in the file.

9.4 Writing Palettes

9.4.1 Writing a Palette: DFPaddpal and DFPputpal

To write a palette to an HDF file, the calling program must contain one of the following function calls:

C: `status = DFPaddpal(filename, palette);`

FORTRAN: `status = dpapal(filename, palette)`

OR

C: `status = DFPputpal(filename, palette, overwrite, filemode);`

FORTRAN: `status = dpppal(filename, palette, overwrite, filemode)`

DFPaddpal and **DFPputpal** will write a palette to an HDF file named by `filename`. When given a new filename, **DFPputpal** and **DFPaddpal** creates a new file and writes the palette as the first object in the file. When given an existing filename, **DFPaddpal** appends the palette to the end of the file.

DFPputpal provides this functionality as well with additional options for how the data is handled, providing more control over how a palette is written to file than **DFPaddpal**. Specifically, the `overwrite` parameter determines whether or not to overwrite the last palette written to a file or to append a new palette onto the file. The `filemode` parameter determines whether to create a new file or to append the data to the previous file. Note the combination to overwrite a palette in a newly created file is invalid and will generate an error. To overwrite a palette, `filename` must be the same filename as the last file accessed through the DFP interface. The parameters for **DFPaddpal** and **DFPputpal** are more explicitly defined in the following table.

TABLE 9B

DFPputpal and DFPaddpal Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFPputpal (dpppal)	filename	char *	character* (*)	Name of the HDF file.
	palette	VOIDP	<valid numeric data type>	768-byte space for palette.
	overwrite	intn	integer	Palette write specification.
	filemode	char *	character* (*)	File write specification.
DFPaddpal (dpapal)	filename	char *	character* (*)	Name of the HDF file.
	palette	VOIDP	<valid numeric data type>	768-byte space with palette.

Calling **DFPaddpal** or **DFPputpal** immediately after writing an 8-bit raster image will not group the palette with the preceding image. Palettes written to a file sequentially can be retrieved sequentially. However, to maintain a higher level of organization between multiple palettes and images stored in the same file, it's a good idea to explicitly group each palette with the image to which it belongs. To find out more about assigning a palette to an image, see Chapter 6, titled *8-bit Raster Images (DFR8 API)*.

EXAMPLE 1.

Writing a Palette

In the following code examples, **DFPaddpal** is used to write a palette to an HDF file named "Example1.hdf".

C:

```
#include "hdf.h"

main( )
{
uint8 palette_data[768];
intn i;
int32 status;

/* Initialize the palette to grayscale. */
for (i = 0; i < 256; i++) {
  palette_data[i * 3] = i;
  palette_data[i * 3 + 1] = i;
  palette_data[i * 3 + 2] = i;
}

/* Write the palette to file. */
```

```
status = DFPaddpal("Example1.hdf", (VOIDP)palette_data);

}
```

FORTRAN:

```
        PROGRAM WRITE PALETTE

         integer dpapal, status, i
         character palette_data(768)

C        Initialize the palette to greyscale.
         do 10, i = 1, 256
           palette_data((i - 1) * 3 + 1) = char(i - 1)
           palette_data((i - 1) * 3 + 2) = char(i - 1)
           palette_data((i - 1) * 3 + 3) = char(i - 1)
10        continue

C        Write the palette to the HDF file.
         status = dpapal('Example1.hdf', palette_data)

         end
```

9.4.2 Specifying the Reference Number of a Palette: DFPwriteref

DFPwriteref specifies the reference number of the palette to be written on the next call to **DFPaddpal** or **DFPputpal:**

C:

```
status = DFPwriteref(filename, ref);
status = DFPaddpal(filename, palette);
```

FORTRAN:

```
status = dpwref(filename, ref)
status = dpapal(filename, palette)
```

DFPwriteref assigns the specified reference number to the next palette written to the file `filename`. If the value of `ref` is the same as the reference number of an existing palette, the existing palette will be overwritten.

The parameters of **DFPwriteref** are further described in the following table.

TABLE 9C **DFPwriteref Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type C	Data Type Fortran-77	Description
DFPwriteref (dpwref)	filename	char *	character* (*)	Name of the HDF file containing the palette.
	ref	uint16	integer	Reference number for the next call to DFPaddpal or DFPputpal.

9.5 Reading a Palette

The DFP programming model for reading a palette is similar to that for writing a palette - only the palette read call is required.

9.5.1 Reading a Palette: DFPgetpal

DFPgetpal is the only function required to read a palette. If the file is being opened for the first time, **DFPgetpal** returns the first palette in the file. Subsequent calls will return successive palettes in the file. In this way palettes are read in the same order in which they were written to the file.

To read a palette from an HDF file, the calling program must contain the following routines:

C: `status = DFPgetpal(filename, palette);`

FORTRAN: `status = dpgpal(filename,palette)`

DFPgetpal retrieves the next palette from the HDF file specified by `filename`. The space allocated for the palette is specified by `palette` and must be at least 768 bytes. When **DFPgetpal** is first called, it returns the first palette in the file. Subsequent calls to **DFPgetpal** will return successive palettes in the order in which they are stored in the file, including those stored via the DFR8 interface.

The parameters of **DFPgetpal** are defined in the following table.

TABLE 9D **DFPgetpal Parameter List**

Function Call (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFPgetpal (**dpapal**)	filename	char *	character* (*)	Name of the HDF file.
	palette	VOIDP	<valid numeric data type>	768-byte buffer for the palette.

EXAMPLE 2. **Reading a Palette**

The following examples demonstrate the method used to read a palette from the "Example1.hdf" HDF file created in Example 1.

C:

```
#include "hdf.h"

main( )
{

uint8 palette_data[768];
intn status;

/* Read the palette data from a file. */
status = DFPgetpal("Example1.hdf", (VOIDP)palette_data);

}
```

FORTRAN:

```
      PROGRAM READ PALETTE

       integer dpgpal, status
       character palette_data(768)

C      Read the palette from the HDF file.
       status = dpgpal('Example1.hdf', palette_data)
```

```
end
```

9.5.2 Reading a Palette with a Given Reference Number: DFPreadref

DFPreadref is used to access specific palettes stored in files containing multiple palettes. It is the optionally called before **DFPgetpal** to set the next palette to be accessed to be the specified palette. **DFPreadref** can be used in connection with vgroups, which identify their members by tag/reference number pair.

To access a specific palette, use the following calling sequence:

C:
```
true_false = DFPreadref(filename, ref);
status = DFPgetpal(filename, palette);
```

FORTRAN:
```
true_false = dprref(filename, ref)
status = dpgpal(filename, palette)
```

DFPreadref specifies the reference number for the next read operation performed on the HDF file `filename` to the reference number specified by `ref`.

The parameters of **DFPreadref** are further defined in the following table.

TABLE 9E **DFPreadref Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFPreadref (dprref)	filename	char *	character* (*)	Name of the HDF file.
	ref	uint16	integer	Reference number of the next palette to be read.

9.5.3 Specifying the Next Palette to be Accessed to be the First Palette: DFPrestart

DFPrestart causes the next **DFPgetpal** to read from the first palette in the file, rather than the palette following the one that was most recently read. **DFPrestart** has the following syntax:

C:
```
status = DFPrestart( );
```

FORTRAN:
```
status = dprest( )
```

9.6 Other Palette Routines

9.6.1 Querying the Number of Palettes in a File: DFPnpals

DFPnpals returns the total number palettes in a file and has the following syntax:

C:
```
num_of_pals = DFPnpals(filename);
```

FORTRAN:
```
num_of_pals = dpnpals(filename)
```

The parameter of **DFPnpals** is further defined in the following table.

TABLE 9F

DFPnpals Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFPnpals (dpnpals)	filename	char *	character* (*)	Name of the HDF file.

9.6.2 Obtaining the Reference Number of the Most Recently Accessed Palette: DFPlastref

DFPlastref returns the reference number most recently used in writing or reading a palette. This routine is used for attaching annotations to palettes and adding palettes to vgroups.

The following calling sequence uses **DFPlastref** to find the reference number of the palette most recently written to an HDF file:

C:

```
status = DFPaddpal(filename, palette, width, height, compress);
lastref = DFPlastref( );
```

FORTRAN:

```
status = dpapal(filename, palette, width, height, compress)
lastref = dplref( )
```

DFPputpal or **DFPgetpal** can be used in place of **DFPaddpal** with similar results.

9.7 Backward Compatibility Issues

As HDF has evolved, a variety of internal structures have been used to store palettes, with different tags used to represent them. To maintain backward compatibility with older versions of HDF, the palette interface supported by HDF version 4.0 recognizes palettes stored using all previously-used HDF tags. A detailed description of the tags and structures used to store palettes is in the *HDF Specification and Developers' Guide*, version 3.2 and later.

Chapter 10

Annotations (DFAN and AN API)

10.1 Chapter Overview

The HDF annotation interface supports the appending of labels and descriptions to HDF files and the data objects they contain. This chapter explains the methods used to read and write file and object annotations. It also explains the alternate multifile annotation interface introduced in version 4.0.

10.2 The Annotation Data Model

When working with different data types, it is often convenient to identify the contents of a file by adding a short text description or ***annotation***. An annotation serves as the label for a file or data element, as in "COLLECTED 12/14/90" or "BLACK HOLE SIMULATION". For example, if the data originated as satellite data, the description might include the source of the data, pertinent environmental conditions, or other relevant information. In the case of a hypothetical black hole simulation, the description might contain source code for the program that produced the data.

As they are implemented as variable-length strings, HDF annotations are designed to accommodate a wide variety of information including titles, comments, variable names, parameters, formulas, and source code. In fact, HDF annotations can encompass any textual information regarding the collection, meaning, or intended use of the data.

10.2.1 Labels and Descriptions

Annotations come in two forms; ***labels*** and ***descriptions***. Labels are short annotations used for assigning things like titles or time stamps to a file or its data objects. Longer annotations are called descriptions, and they typically contain more extensive information such as a source code module or mathematical formulae.

Labels are defined as a null-terminated string of characters except `NULL`. Descriptions may contain any sequence of ASCII characters including `NULL`.

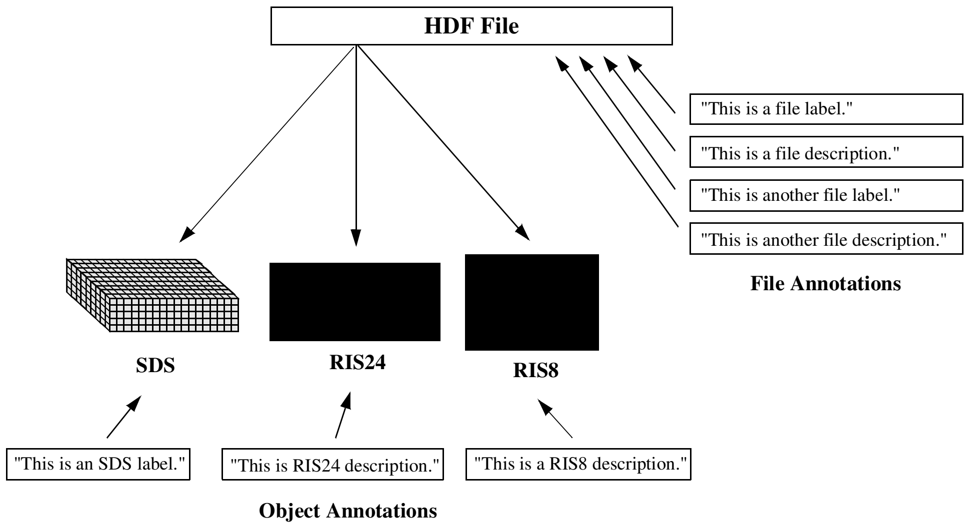
In addition to the distinction made between labels and descriptions, HDF distinguishes between ***file annotations*** and ***object annotations***.

10.2.2 File Annotations

File annotations are text strings which describe the origin, meaning, or intended use of its data. Any HDF file can be annotated with a label, description, or combination of both. (See Figure 10a.) The number of labels or descriptions an HDF file may contain is limited to the maximum number of tag/reference number pairs. File annotations may be assigned in any order and at any time after a file is created.

FIGURE 10a

File and Object Annotations



Although it may seem convenient to use a file annotation to describe one data object in a file, this practice is not recommended because any additional objects added to the file will obscure the implicit link between the original object and its annotation.

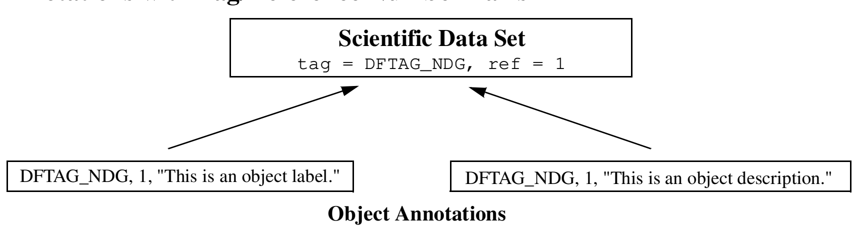
10.2.3 Object Annotations

Object annotations are text strings assigned to individual data objects to explain their origin, meaning, or intended use. Because object annotations are assigned to individual objects, their use requires an understanding of HDF tags and reference numbers.

The annotation interface takes advantage of this identification scheme by including the object's tag/reference number pair with the text of the annotation. Consider a scientific data set identified by the tag `DFTAG_NDG` and the reference number `1`. (See Figure 10b.) All object annotations assigned to this particular data set must be prefaced with the tag `DFTAG_NDG` followed by the reference number `1`.

FIGURE 10b

Object Annotations with Tag/Reference Number Pairs



10.3 The Single-File Annotation Interface

The functions and routines that comprise the single-file annotation interface have names that begin with the string "DFAN" and the equivalent Fortran-77 routine names are prefaced by "da".This interface is the older annotation interface and only supports annotation access within one particular HDF file. It doesn't support the concept of an annotation identifier used in the newer multifile interface, therefore annotations created with the multifile interface cannot be accessed or manipulated with single-file interface functions. The multifile interface is described in Section 10.9 on page 264.

DFANaddfid, **DFANgetfid**, **DFANaddfds**, and **DFANgetfds** are the functions used for annotating files and are sufficient for most read and write operations.

10.3.1 Single-File Annotation Library Routines

These functions are divided into the following categories:

- ***Write functions*** assign a file or object annotation.
- ***Read functions*** retrieve a file or object annotation.
- ***General inquiry functions*** return a list of all labels and reference numbers.

The DFAN function calls are defined in the following table and on their respective pages in the *HDF Reference Guide*.

TABLE 10A **DFAN Library Routines**

Purpose	Functions		Description
	C	Fortran-77	
Write	DFANaddfid	daafid	Assigns a file label to a specific file.
	DFANaddfds	daafds	Assigns a file description to a specific file.
	DFANputlabel	daplab	Assigns an object label to a specific data object.
	DFANputdesc	dapdesc	Assigns an object description to a specific data object.
Read	DFANgetfidlen	dagfidl	Returns the length of a file label.
	DFANgetfid	dagfid	Reads the text of a file label.
	DFANgetfdslen	dagfdsl	Returns the length of a file description.
	DFANgetfds	dagfds	Reads the text of a file description.
	DFANgetlablen	dagllen	Returns the length of an object label.
	DFANgetlabel	daglab	Reads the text of an object label.
	DFANgetdesclen	dagdlen	Returns the length of an object description.
	DFANgetdesc	dagdesc	Reads the text of an object description.
General Inquiry	DFANlablist	dallist	Gets a list of all the labels in a file for a particular tag.
	DFANlastref	dalref	Returns the reference number of the last annotation accessed.
Maintenance	DFANclear	None	Clear the internal tables and strcutures used by the DFAN interface.

10.3.2 Tags in the Annotation Interface

See Table 10B below for a list of the annotation tags defined in HDF versions 2.0, 3.0 and 4.0. Newly-defined tags names in each version are bolded. For a more complete list of tags consult the *HDF Specification and Developer's Guide*.

TABLE 10B **List of Annotation Interface Tags in HDF Versions 2.0, 3.0 and 4.0**

Interface	Data Object	Tag Name		
		v2.0	v3.0	v4.0

DFR8	Raster Image: 8-bit (uncompressed)	DFTAG_RI8	DFTAG_RI	DFTAG_RI
	Compressed Image: 8-bit	DFTAG_CI8	DFTAG_CI	DFTAG_CI
	Image Dimension: 8-bit	DFTAG_ID8	DFTAG_ID	DFTAG_ID
	Image Palette: 8-bit	DFTAG_IP8	DFTAG_LUT	DFTAG_LUT
DF24	Raster Image Group	None	DFTAG_RIG	DFTAG_RIG
	Raster Image (uncompressed)	None	DFTAG_RI	DFTAG_RI
	Compressed Image	None	DFTAG_CI	DFTAG_CI
	Image Dimension	None	DFTAG_ID	DFTAG_ID
DFP	Color Look-up Table	DFTAG_LUT	DFTAG_LUT	DFTAG_LUT
DFSD	Scientific Data Group	DFTAG_SDG	DFTAG_SDG	DFTAG_NDG
	Scientific Data	DFTAG_SD	DFTAG_SD	DFTAG_SD
	Scientific Data Dimension	DFTAG_SDD	DFTAG_SDD	DFTAG_SDD
	Scientific Data Scale Attribute	DFTAG_SDS	DFTAG_SDS	DFTAG_SDS
	Scientific Data Label Attribute	DFTAG_SDL	DFTAG_SDL	DFTAG_SDL
	Scientific Data Unit Attribute	DFTAG_SDU	DFTAG_SDU	DFTAG_SDU
	Scientific Data Format Attribute	DFTAG_SDF	DFTAG_SDF	DFTAG_SDF
	Scientific Data Max/Min Attribute	DFTAG_SDM	DFTAG_SDM	DFTAG_SDM
	Scientific Data Coordinates Attribute	DFTAG_SDC	DFTAG_SDC	DFTAG_SDC
DFAN	File Identifier	DFTAG_FID	DFTAG_FID	DFTAG_FID
	File Descriptor	DFTAG_FD	DFTAG_FD	DFTAG_FD
	Data Identifier Label	DFTAG_DIL	DFTAG_DIL	DFTAG_DIL
	Data Identifier Annotation	DFTAG_DIA	DFTAG_DIA	DFTAG_DIA
Vdata	Vdata Storage	DFTAG_VS	DFTAG_VS	DFTAG_VS
Vgroups	Vgroup Storage	DFTAG_VG	DFTAG_VG	DFTAG_VG

10.4 Programming Model for the Annotation Interface

There are two general programming models for the DFAN interface. The first programming model addresses file annotation, and the second addresses object annotation. In the case of file annotations, the DFAN interface relies on the calling program to initiate and terminate access to files. This approach necessitates the following programming model:

1. Open the file.
2. Perform the desired file annotation operation.
3. Close the file.

The object annotation programming model is a simplified version of the file annotation programming model:

1. Perform the desired object annotation operation.

Essentially, the difference between the two models is that file annotations require **Hopen** and **Hclose** to open and close the target files whereas object annotations do not.

10.5 Writing Annotations

The DFAN interface supports writes to file labels, file descriptions, object labels and object descriptions.

10.5.1 Assigning a File Label: DFANaddfid

To write a file label, the calling program must call **DFANaddfid**:

C: `status = DFANaddfid(file_id, label);`

FORTRAN: `status = daafid(file_id, label)`

DFANaddfid has two parameters: `file_id` and `label`. The `file_id` parameter contains the file identifier for the file to be annotated and the `label` parameter contains the annotation string. The label array must be null-terminated. In the Fortran-77 version, the length of the label should be the length of the label array as in Fortran-77 string lengths are assumed to be the declared length of the array that holds the string.

The parameters of **DFANaddfid** is further defined below. (See Table 10F on page 259.)

10.5.2 Assigning a File Description: DFANaddfds

To write a file description, the calling program must call **DFANaddfds**:

C: `status = DFANaddfds(file_id, description, desc_length);`

FORTRAN: `status = daafds(file_id, description, desc_length)`

DFANaddfds has three parameters: `file_id`, `description`, and `desc_length`. The `file_id` parameter contains the file identifier and the `description` parameter contains the label string. The parameter description can contain any sequence of ASCII characters and is not limited to a single string. The `desc_length` parameter specifies the length of the annotation.

The parameters of **DFANaddfds** are defined in the following table.

TABLE 10C **DFANaddfid and DFANaddfds Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFANaddfid (daafid)	file_id	int32	integer	File identifier.
	label	char *	character* (*)	File label string.
DFANaddfds (daafds)	file_id	int32	integer	File identifier.
	description	char *	character* (*)	File description string.
	desc_length	int32	integer	Length of the description in bytes.

EXAMPLE 3. **Writing a File Label and a File Description**

The following examples add a file label and description to the file named "Example1.hdf". Notice that after the file is opened, the `file_id` may be used to add any combination of file annotations before the file is closed.

C:

```
#include "hdf.h"
```

```
main( )
{

int32 file_id;
intn status;
static char file_label[] = "This is a file label.";
static char file_desc[] = "This is a file description.";

/* Open the HDF file to write the annotations. */
file_id = Hopen("Example1.hdf", DFACC_CREATE, 0);

/* Write the label to the file. */
status = DFANaddfid(file_id, file_label);

/* Write the description to the file. */
status = DFANaddfds(file_id, file_desc, strlen(file_desc));

/* Close the file. */
status = Hclose(file_id);

}
```

FORTRAN:

```
       PROGRAM CREATE ANNOTATION

        character*50 file_label, file_desc
        integer daafid, daafds, status, file_id, hopen, hclose

        integer*4 DFACC_CREATE
        parameter (DFACC_CREATE = 4)

        file_label = "This is a file label."
        file_desc = "This is a file description."

C       Open the HDF file to write the annotations.
        file_id = hopen('Example1.hdf', DFACC_CREATE, 0)

C       Write the label to the file.
        status = daafid(file_id, file_label)

C       Write the description to the file.
        status = daafds(file_id, file_desc, 26)

C       Close the file.
        status = hclose(file_id)

        end
```

10.5.3 Assigning an Object Label: DFANputlabel

To write a file label, the calling program must contain a call to **DFANputlabel**:

C: `status = DFANputlabel(filename, tag, ref, label);`

FORTRAN: `status = daplab(filename, tag, ref, label)`

DFANputlabel has four parameters: `filename`, `tag`, `ref`, and `label`. The `label` parameter contains a single null-terminated string that defines the annotation.

The parameters of **DFANputlabel** are further defined in Table 10F below.

10.5.4 Assigning an Object Description: DFANputdesc

To write an object description, the calling program must contain a call to **DFANputdesc**:

C:
```
status = DFANputdesc(filename, tag, ref, description, desc_len);
```

FORTRAN:
```
status = dapdesc(filename, tag, ref, description, desc_len)
```

DFANputdesc has five parameters: `filename`, `tag`, `ref`, `description`, and `desc_len`. The `filename` parameter is the name of the HDF file containing the object to be annotated. The `tag` and `ref` parameters are the tag/reference number pair of the object to be annotated. The `description` parameter contains a buffer for the annotation text and the `desc_len` parameter specifies the length of the buffer.

The parameters of **DFANputdesc** are further defined in the following table.

TABLE 10D **DFANputlabel and DFANputdesc Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type C	Data Type Fortran-77	Description
DFANputlabel (daplab)	filename	char *	character* (*)	Name of the file to be accessed.
	tag	uint16	integer	Tag of the object to be annotated.
	ref	uint16	integer	Reference number of the object to be annotated.
	label	char *	character* (*)	Object label string.
DFANputdesc (dapdesc)	filename	char *	character* (*)	Name of the file to be accessed.
	tag	uint16	integer	Tag of the object to be annotated.
	ref	uint16	integer	Reference number of the object to be annotated.
	label	char *	character* (*)	Object description string.
	desc_len	int32	integer	Length of the description in bytes.

EXAMPLE 4. **Writing an Object Label and Description to a Scientific Data Set**

These examples illustrate the use of **DFANputlabel** and **DFANputdesc** to assign both an object label and an object description to a scientific data set immediately after it is written to file. The tag for scientific data sets is `DFTAG_NDG`.

C:

```
#include "hdf.h"

#define X_LENGTH 3
#define Y_LENGTH 2
#define Z_LENGTH 5

main( )
{

/* Create the data array. */
static float32 sds_data[X_LENGTH][Y_LENGTH][Z_LENGTH] =
{  1,  2,  3,  4,  5,
   6,  7,  8,  9, 10,
```

```
  11, 12, 13, 14, 15,
  16, 17, 18, 19, 20,
  21, 22, 23, 24, 25,
  26, 27, 28, 29, 30 };

/*
* Create the array that will hold the dimensions of
* the data array.
*/
int32 dims[3] = {X_LENGTH, Y_LENGTH, Z_LENGTH};
intn refnum, status;
static char object_desc[] = "This is an object description.";
static char object_label[] = "This is an object label.";

/* Write the data to the HDF file. */
status = DFSDadddata("Example1.hdf", 3, dims, (VOIDP)sds_data);

/* Get the reference number for the newly written data set. */
refnum = DFSDlastref( );

/* Assign the object label to the scientific data set. */
status = DFANputlabel("Example1.hdf", DFTAG_NDG, refnum, \
                object_label);

/* Assign the object description to the scientific data set. */
status = DFANputdesc("Example1.hdf", DFTAG_NDG, refnum, \
                object_desc, strlen(object_desc));

}
```

FORTRAN:

```
     PROGRAM ANNOTATE OBJECT

      integer dsadata, dims(3), status, refnum
      integer daplab, dapdesc, dslref

      integer*4 DFTAG_NDG, X_LENGTH, Y_LENGTH, Z_LENGTH
      parameter(DFTAG_NDG = 720,
     +          X_LENGTH = 5,
     +          Y_LENGTH = 2,
     +          Z_LENGTH = 3)

C     Create the data array.
      real*4 sds_data(X_LENGTH, Y_LENGTH, Z_LENGTH)
      data sds_data /
     +          1,  2,  3,  4,  5,
     +          6,  7,  8,  9, 10,
     +         11, 12, 13, 14, 15,
     +         16, 17, 18, 19, 20,
     +         21, 22, 23, 24, 25,
     +         26, 27, 28, 29, 30  /

C     Create the array the will hold the dimensions of the data array.
      data dims /X_LENGTH, Y_LENGTH, Z_LENGTH/

C     Write the data to the HDF file.
      ref = dsadata('Example1.hdf', 3, dims, sds_data)

C     Get the reference number for the newly written data set.
      refnum = dslref( )

C     Assign the object label to the scientific data set.
```

```
      status = daplab('Example1.hdf', DFTAG_NDG, refnum,
     +            'This is an object label.')

C     Assign an object description to the scientific data set.
      status = dapdesc('Example1.hdf', DFTAG_NDG, refnum,
     +             'This is an object description.', 30)

      end
```

10.6 Reading Annotations

The DFAN interface provides two functions for reading file annotations, which are described below.

10.6.1 Reading a File Label: DFANgetfidlen and DFANgetfid

The DFAN programming model for reading a file label is as follows:

1. Get the length of the label.
2. Read the file label.

To read the first file label in a file, the calling program must contain the following function calls:

C:

```
isfirst = 1;
label_length = DFANgetfidlen(file_id, isfirst);
label_buffer = HDgetspace(label_length);
fid_len = DFANgetfid(file_id, label_buffer, label_length,
                            isfirst);
```

FORTRAN:

```
isfirst = 1
label_length = dagfidl(file_id, isfirst)
fid_len = dagfid(file_id, label_buffer, label_length, isfirst)
```

DFANgetfidlen has two parameters: `file_id` and `isfirst`. The `isfirst` parameter specifies whether the first or subsequent file annotations are to be read. To read the first file label length `isfirst` should be set to the value `1`, to sequentially step through all the remaining file labels assigned to a file `isfirst` should be set to `0`.

When **DFANgetfidlen** is first called for a given file, it returns the length of the first file label. To get the lengths of subsequent file labels, you must call **DFANgetfid** between calls to **DFANgetfidlen**. Otherwise, additional calls to **DFANgetfidlen** will return the length of the same file label.

DFANgetfid has four parameters: `file_id`, `label_buffer`, `label_length`, and `isfirst`. The `label_buffer` parameter is a pointer to a buffer for the label text. The `label_length` parameter is the length of the buffer in memory which can be shorter than the full length of the label in the file. If the `label_length` is not large enough, the label is truncated to `label_length-1` characters in the buffer `label_buffer`. The `isfirst` parameter is used to read the first or subsequent file annotations. To read the first file label `isfirst` should be set to `1`, to sequentially step through all the remaining file labels assigned to a file `isfirst` should be set to `0`.

HDgetspace is described in Chapter 2, titled *HDF Fundamentals*.

The parameters of **DFANgetfidlen** and **DFANgetfid** are described below. (See Table 10E on page 256.)

10.6.2 Reading a File Description: DFANgetfdslen and DFANgetfds

The DFAN programming model for reading a file description is as follows:

1. Get the length of the description.
2. Read the file description.

To read the first file description in a file, the calling program must contain the following calls:

C:
```
isfirst = 1;
desc_length = DFANgetfdslen(file_id, isfirst);
desc_buffer = HDgetspace(desc_length);
fds_len = DFANgetfds(file_id, desc_buf, desc_length, isfirst);
```

FORTRAN:
```
isfirst = 1
desc_length = dagfdsl(file_id, isfirst)
fds_len = dagfds(file_id, desc_buf, desc_length, isfirst)
```

DFANgetfdslen has two parameters: `file_id` and `isfirst`. The `isfirst` parameter specifies whether the first or subsequent file annotations are to be read. To read the first file description length `isfirst` should be set to the value `1`, to sequentially step through all the remaining file descriptions assigned to a file `isfirst` should be set to `0`.

When **DFANgetfdslen** is first called for a given file, it returns the length of the first file description. To get the lengths of successive file descriptions, you must call **DFANgetfds** between calls to **DFANgetfdslen** as with **DFANfidgetlen**.

DFANgetfds has four parameters: `file_id`, `desc_buf`, `desc_length`, and `isfirst`. The `desc_buffer` parameter is a pointer to a buffer for the description text. The `desc_length` parameter is the length of the buffer in memory which can be shorter than the full length of the description in the file. If the `desc_length` is not large enough, the description is truncated to `desc_length` characters in the buffer `desc_buf`. The `isfirst` parameter specifies whether the first or subsequent file annotations are to be read. To read the first file description `isfirst` should be set to the value `1`, to sequentially step through all the remaining file descriptions assigned to a file `isfirst` should be set to `0`.

The parameters of these routines are described further in the following table.

TABLE 10E **DFANgetfidlen, DFANgetfid, DFANgetfdslen and DFANgetfds Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFANgetfidlen (dagfidl)	file_id	int32	integer	File identifier.
	isfirst	intn	integer	Location of the next annotation.
DFANgetfid (dagfid)	file_id	int32	integer	File identifier.
	desc_buf	char *	character* (*)	File label buffer.
	buf_length	int32	integer	Label buffer length.
	isfirst	intn	integer	Location of the next annotation
DFANgetfdslen (dagfdsl)	file_id	int32	integer	File identifier.
	isfirst	intn	integer	Location of the next annotation.
DFANgetfds (dagfds)	file_id	int32	integer	File identifier.
	description	char *	character* (*)	File description buffer.
	desc_length	int32	integer	Description buffer length.
	isfirst	intn	integer	Location of the next annotation.

EXAMPLE 5.

Reading a File Label and a File Description

The following examples read a file label from the HDF file named "Example1.hdf". The **DFANgetfidlen** routine is used to verify the length of the label before the read operation is performed. The argument "1" in both routines indicate the first description in the HDF file is the target. **DFANgetfdslen** and **DFANgetfds** can be directly substituted for **DFANgetfidlen** and **DFANgetfid** in order to read a file description instead of a file label.

C:

```
#include "hdf.h"

main( )
{
int32 file_id, file_label_len;
char *file_label;
intn status;

/* Open the HDF file containing the annotation. */
file_id = Hopen("Example1.hdf", DFACC_READ, 0);

/* Determine the length of the file label. */
file_label_len = DFANgetfidlen(file_id, 1);

/* Allocated memory for the file label buffer. */
file_label = HDgetspace(file_label_len);

/* Read the file label. */
file_label_len = DFANgetfid(file_id, file_label, file_label_len, 1);

/* Close the file */
status = Hclose(file_id);

}
```

FORTRAN:

```
     PROGRAM GET ANNOTATION

      integer status, file_id, label_length
      integer hopen, hclose, dagfidl, dagfid
      character file_label(50)

      integer*4 DFACC_READ
      parameter(DFACC_READ = 1)

C     Open the HDF file containing the file label.
      file_id = hopen("Example1.hdf", DFACC_READ, 0)

C     Determine the length of the file label.
      label_length = dagfidl(file_id, 1)

C     Read the file label.
      status = dagfid(file_id, file_label, label_length, 1)

C     Close the HDF file.
      status = hclose(file_id)

      end
```

10.6.3 Reading an Object Label: DFANgetlablen and DFANgetlabel

The DFAN programming model for reading a object label is as follows:

1. Get the length of the label.
2. Read the file label.

To read the first object label in a file, the calling program must contain the following routines:

C:
```
label_length = DFANgetlablen(filename, tag, ref);
label_buf = HDgetspace(label_length);
status = DFANgetlabel(filename, tag, ref, label_buf,
                      label_length);
```

FORTRAN:
```
label_length = daglabl(filename, tag, ref)
status = daglab(filename, tag, ref, label_buf, label_length)
```

DFANgetlablen returns the length of the label assigned to the object identified by the given tag/reference number pair. **DFANgetlabel** must be called between calls to **DFANgetlablen** - it is the routine that actually returns the label and prepares the API to read the next label.

DFANgetlabel has five parameters: `filename`, `tag`, `ref`, `label_buf`, and `label_length`. The `label_buf` parameter is a pointer to a buffer that stores the label text. The `label_length` parameter is the length of the buffer in memory - this can be shorter than the full length of the label in the file. If the `label_length` is not large enough, the label is truncated to `label_length` characters in the buffer `label_buf`. The length of `label_buf` must be at least one greater than the anticipated length of the label, to account for the null termination appended to the label text.

The parameters of **DFANgetlablen** and **DFANgetlabel** are defined below.

10.6.4 Reading an Object Description: DFANgetdesclen and DFANgetdesc

The DFAN programming model for reading a object description is as follows:

1. Get the length of the description.
2. Read the file description.

To read the first object description in a file, the calling program must contain the following routines:

C:
```
desc_length = DFANgetdesclen(filename, tag, ref);
desc_buf = HDgetspace(desc_length);
status = DFANgetdesc(filename, tag, ref, desc_buf, desc_length);
```

FORTRAN:
```
label_length = dagdlen(filename, tag, ref)
status = dagdesc(filename, tag, ref, desc_buf, desc_length)
```

DFANgetdesclen returns the length of the description assigned to the object identified by the specified tag/reference number pair. **DFANgetdesc** must be called between calls to **DFANgetdesclen** to reset the current object description to the next in the file.

DFANgetdesc takes five parameters: `filename`, `tag`, `ref`, `desc_buf`, and `desc_length`. The `desc_buf` parameter is a pointer to the buffer that stores the description text. The `desc_length` parameter is the length of the buffer in memory which can be shorter than the full length of the description in the file. If the `desc_length` is not large enough, the description is truncated to `desc_length` characters in the buffer `desc_buf`.

The parameters of **DFANgetdesclen** and **DFANgetdesc** are defined in the following table.

TABLE 10F

DFANgetlablen, DFANgetlabel, DFANgetdesc and DFANgetdesclen Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFANgetlablen (dagllen)	filename	char *	character* (*)	Name of the file to be accessed.
	tag	uint16	integer	Tag assigned to the annotated object.
	ref	uint16	integer	Reference number for the annotated object.
DFANgetlabel (daglab)	filename	char *	character* (*)	Name of the file to be accessed.
	tag	uint16	integer	Tag assigned to the annotated object.
	ref	uint16	integer	Reference number assigned to the annotated object.
	label_buf	char *	character* (*)	Buffer for the returned annotation.
	label_length	int32	integer	Size of the buffer allocated to hold the annotation.
DFANgetdesclen (dagdlen)	filename	char *	character* (*)	Name of the file to be accessed.
	tag	uint16	integer	Tag assigned to the annotated object.
	ref	uint16	integer	Reference number for the annotated object.
DFANgetdesc (dagdesc)	filename	char *	character* (*)	Name of the file to be accessed.
	tag	uint16	integer	Tag assigned to the annotated object.
	ref	uint16	integer	Reference number assigned to the annotated object.
	desc_buf	char *	character* (*)	Buffer for the returned annotation.
	desc_length	int32	integer	Size of the buffer allocated to hold the annotation.

EXAMPLE 6.

Reading an Object Label and Description

The following examples demonstrate the use of **DFANgetdesclen** and **DFANgetdesc** to read an object description assigned to a scientific data set. These examples assume that, in addition to other data objects, the "Example1.hdf" HDF file also contains multiple scientific data sets, some of which may not be annotated. **Hfind** is used to determine the reference number for the first annotated scientific data object in the file.

C:

```
#include "hdf.h"

main( )
{
intn desc_length = -1, status;
char desc[50];
int32 file_id;
uint16 tag = 0, ref = 0;
uint32 find_offset, find_length;

/* Open the file and initialize the searching parameters to 0. */
file_id = Hopen("Example1.hdf", DFACC_READ, 0);

/*
* Start a sequential forward search for the first reference
* number assigned to a scientific data set.
*/
while (Hfind(file_id, DFTAG_NDG, DFREF_WILDCARD, &tag, &ref, \
    &find_offset, &find_length, DF_FORWARD) != FAIL) {

/*
```

```
* After discovering a valid reference number, check for an
* object description by returning the length of the description.
* If the inquiry fails, continue searching for the next valid
* reference number assigned to a scientific data set.
*/
if ((desc_length = DFANgetdesclen("Example1.hdf", tag, ref)) \
    == FAIL)
    break;

/*
* If a description exists and it will fit in the description buffer,
* print it.
*/
if (desc_length != FAIL && desc_length <= 50) {
    status = DFANgetdesc("Example1.hdf", tag, ref, desc, desc_length);
    printf("Description: %s\n", desc);
}
}

/* Close the file. */
status = Hclose(file_id);

}
```

FORTRAN:

There is no Fortran-77 version of the Example 4 C code for this version of the documentation as there is no Fortran-77 equivalent of **Hfind**.

10.7 Maintenance Routines

The DFAN interface provides only function for interface maintenance - **DFANclear**.

10.7.1 Clearing the DFAN Interface's Internal Structures and Settings: DFANclear

DFANclear clears all internal library structures and parameters of the DFAN annotiation interface.

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

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

The parameters are described further in the following table.

TABLE 10G **DFANclear Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFANclear **(daclear)**	None	None	None	None.

10.8 Determining Reference Numbers

It is advisable to check the reference number before attempting to assign an object annotation, as the overwriting of reference numbers isn't prevented by the HDF library routines.

There are three ways to check a reference number for an object:

- Access the object with a read or write operation followed by **DF*lastref**.
- Call **DFANlablist** to return a list of all assigned reference numbers for a given tag.
- Call **Hfind** to locate an object with a given tag/reference number pair.

10.8.1 Determine a Reference Number for the Last Object Accessed: DF*lastref

There are two methods of obtaining a reference number through the use of a **DF*lastref** call. The first approach is to obtain and store the reference number of an object immediately after the object is created:

1. Create the data object.
2. Call **DF*lastref** to determine its reference number.
3. Read or write an object annotation.

The second approach is to determine the reference number at some time after the data object is created. This approach requires repeated **DF*read** calls until the appropriate object is accessed, followed by a call to **DF*lastref**:

1. Read the appropriate data object.
2. Call **DF*lastref** to determine its reference number.
3. Read or write and object annotation.

Most HDF interfaces provide one routine that assigns a specified reference number to a data object and another routine that returns the reference number for the last data object accessed. (See Table 10H.) However, the SD interface doesn't. Also, the DFAN annotation doesn't include a **DF*lastref** routine.

Although **DF*writeref** calls are designed to assign specific reference numbers, they are not recommended for general use because there is no protection against reassigning an existing reference number and overwriting data. In general, it is better to determine a reference number for a data object by calling **DF*lastref** immediately after reading or writing a data object.

The **DF*lastref** routines have no parameters. The **DF*writeref** routines have two: `filename`, which is the name of the file that contains the data object, and `ref`, which is the reference number for the next data object read operation.

TABLE 10H **List and Descriptions of the DF*writeref and DF*lastref Routines**

HDF Data Object	Routine Name (Fortran-77)	Description
8-bit Raster Image	DFR8writeref (d8wref)	Assigns the specified number as the reference number for the next 8-bit raster write operation and updates the write counter to the reflect highest reference number.
	DFR8lastref (d8lref)	Returns the reference number for the last 8-bit raster image set accessed.

HDF Data Object	Routine Name (Fortran-77)	Description
24-bit Raster Image	**DF24writeref (d2wref)**	Assigns the specified number as the reference number for the next 24-bit raster write operation and updates the write counter to reflect the highest reference number.
	DF24lastref (d2lref)	Returns the reference number for the last 24-bit raster image set accessed.
Palette	**DFPwriteref (dpwref)**	Assigns the specified number as the reference number for the next palette write operation and updates the write counter to reflect the highest reference number.
	DFPlastref (dplref)	Returns the reference number for the last palette accessed.
DFSD Scientific Data	**DFSDwriteref (dswref)**	Assigns the specified number as the reference number for the next SDS write operation and updates the write counter to reflect the highest reference number.
	DFSDlastref (dslref)	Returns the reference number for the last scientific data set accessed.
Annotation	**DFANlastref (dalref)**	Returns the reference number for the last annotation accessed.

10.8.2 Querying a List of Reference Numbers for a Given Tag: DFANlablist

Given a tag and two buffers, **DFANlablist** will fill one buffer with all reference numbers for the given tag and the other with all labels assigned to the given tag. The programming model for determining a list of reference numbers is as follows:

1. Determine the number of reference numbers that exist for a given tag.
2. Allocate a buffer to store the reference numbers.
3. Specify the maximum label length.
4. Allocate a buffer to store the labels.
5. Store the list of reference numbers and their labels.

To create a list of reference numbers and their labels for a given tag, the following routines should be called:

C:

```
num_refs = Hnumber(file_id, tag);
ref_buf = HDmalloc(sizeof(uint16*)*num_refs);
max_lab_len = 16;
label_buf = HDmalloc(max_lab_len * num_refs);
start_pos = 0;
num_of_refs = DFANlablist(filename, tag, ref_buf, label_buf,
                                    num_refs, max_lab_len,
                                    start_pos);
```

FORTRAN:

```
num_refs = hnumber(file_id, tag)
max_lab_len = 16
start_pos = 0
num_of_refs = dallist(filename, tag, ref_buf, label_buf,
                             num_refs, max_lab_len, start_pos)
```

Hnumber determines how many objects with the specified tag are in a file. It is described in Chapter 2, titled *HDF Fundamentals*.

DFANlablist has seven parameters: `filename`, `tag`, `ref_list`, `label_buf`, `num_refs`, `max_lab_len`, and `start_pos`. The `filename` parameter specifies the name of the file to search and `tag` specifies the search tag to use when creating the reference and label list. The `ref_buf` and `label_buf` parameters are buffers used to store the reference numbers and labels associated with `tag`. The `num_ref` parameter specifies the length of the reference number list and the `max_lab_len` parameter specifies the maximum length of a label. The `start_pos` parameter specifies the first label to read. For instance, if `start_pos` has a value of `1` all labels will be read - if it has a value of `4`, all but the first three labels will be read.

Taken together, the `ref_list` and `label_list` constitute a directory of all objects and their labels for a given tag. The contents of `label_list` can be displayed to show 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 direct access to data objects in HDF files.

TABLE 10I

DFANlablist Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFANlablist (dallist)	filename	char *	character* (*)	Name of the file to be accessed.
	tag	uint16	integer	Tag assigned to the annotated object.
	ref_list	uint16 []	integer (*)	Reference number for the annotated object.
	label_list	char *	character* (*)	Buffer for the labels.
	list_len	int	integer	Size of the reference number and label lists.
	label_len	intn	integer	Maximum label length.
	start_pos	intn	integer	First entry in the reference number and label lists to be returned.

EXAMPLE 7.

Getting a List of Labels for All Scientific Data Sets

These examples illustrate the method used to get a list of all labels used in scientific data sets in an HDF file using **DFANlablist.** The DFS_MAXLEN definition is located in the "hlimits.h" include file.

C:

```
#include "hdf.h"

#define LISTSIZE 20

main( )
{

int i, num_of_labels, start_position = 1, list_length = 10;
uint16 ref_list[LISTSIZE];
char label_list[DFS_MAXLEN*LISTSIZE-1];

/* Get the total number of labels in the "Example1.hdf" file. */
num_of_labels = DFANlablist("Example1.hdf", DFTAG_NDG, ref_list, \
          label_list, list_length, DFS_MAXLEN,   \
          start_position);

/*
* Print the reference numbers and label names for each label
* in the list.
*/
for (i = 0; i < num_of_labels; i++)
  printf("\n\t%d\tRef number: %d\tLabel: %s", i+1, ref_list[i], \
         label_list - (i * 13));
```

```
printf("\n");

}
```

FORTRAN:

```
      PROGRAM GET LABEL LIST

       integer dallist
       integer*4 DFTAG_NDG, LISTSIZE, DFS_MAXLEN

       parameter (DFTAG_NDG = 720,
      +          LISTSIZE = 20,
      +          DFS_MAXLEN = 255)

       character*60 label_list(DFS_MAXLEN*LISTSIZE)
       integer i, num_of_labels, start_position, ref_list(DFS_MAXLEN)

       start_position = 1

       num_of_labels = dallist('Example1.hdf', DFTAG_NDG, ref_list,
      +                       label_list, 10, DFS_MAXLEN,
      +                       start_position)

       do 10 i = 1, num_of_labels
         print *,'    Ref number:  ',ref_list(i),
      +          '    Label: ',label_list(i)
10     continue

       end
```

10.8.3 Locate an Object by Its Tag and Reference Number: Hfind

Instead of using **DFANlablist** to create a list of reference numbers to search, HDF provides a general search routine called **Hfind**. **Hfind** is described in Chapter 2, titled *HDF Fundamentals*.

10.9 The Multifile Annotation Interface

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". 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 rather than requiring the program to deal with the annotations on a file-by-file basis. The design of the AN interface is similar to the multifile interfaces for raster image (GR) and scientific data set objects (SD).

10.9.1 AN Library Routines

These functions are divided into the following categories:

- ***Access routines*** initialize the interface and provide and terminate access to annotations.
- ***Read/write functions*** read and write file or object annotations.
- ***General inquiry functions*** return information about the annotations in a file.

The AN function calls are defined in the following table and on their respective pages in the *HDF Reference Guide*.

TABLE 10J **AN Library Routines**

Category	Routine Names		Description
	C	Fortran-77	
Access	ANstart	afstart	Initialize AN interface and open file.
	ANselect	afselect	Gets an annotation identifier from a specified index.
	ANend	afend	Close file and AN interface.
	ANcreate	afcreate	Create a new data annotation.
	ANcreatef	affcreate	Create a new file annotation.
	ANendaccess	afendaccess	Terminate access to the annotation that corresponds to the given annotation identifier.
Read/write	ANwriteann	afwriteann	Write the annotation the corresponds to the given annotation identifier.
	ANreadann	afreadeann	Read the annotation the corresponds to the given annotation identifier.
General Inquiry	ANnumann	afnumann	Get the number of annotations in the file that correspond to a given type and tag/reference number pair.
	ANannlist	afannlist	Get a list of annotations in the file that correspond to a given type and tag/reference number pair.
	ANannlen	anannlen	Get the length of the annotation corresponding to a given annotation identifier.
	ANfileinfo	anfileinfo	Get information concerning the number of annotations of each type in the file.
	ANget_tagref	afgettagref	Get a tag/reference number pair from a specified annotation type and index.
	ANid2tagref	afidtagref	Get a tag/reference number pair from a specified annotation id.
	ANtagref2id	aftagrefid	Get an annotation id from a specified tag/reference number pair.
	ANatype2tag	afatypetag	Get an object tag corresponding to a specified annotation type.
	ANtag2atype	aftagatype	Get an annotation type corresponding to a specified object tag.

10.9.2 Type Definitions Used in the Multifile Annotation Interface

The multifile annotation interface uses the four general annotation types used in HDF: the data label, the data description, the file label and the file description. These annotation types correspondingly map to the `AN_DATA_LABEL`, the `AN_DATA_DESC`, the `AN_FILE_LABEL` and the `AN_FILE_DESC` definitions. Several routines in the multifile annotation interface require one of these type definitions to be passed in as an argument to designate the kind of annotation to be created or accessed.

10.10 Programming Model for the Multifile Annotation Interface

The programming model for the multifile annotation interface is as follows:

1. Open a file.
2. Initialize the multifile annotation interface and obtain an interface id.
3. Create or obtain an annotation id.
4. Perform the desired annotation operation.
5. Terminate access to the annotation interface and the annotation.
6. Close the file.

As with the GR and SD interfaces an identifier to the object to be created or accessed is obtained when the interface is initialized. This programming model allows several files to be opened and the contents operated on as long as the calling program accurately keeps track of each file, interface and object identifier returned by the interface. Each object and file must be explicitly closed before the termination of the calling program.

10.11 Creating and Writing Annotations Using the AN Interface

The AN interface writes file labels, file descriptions, data object labels and data object descriptions according to the general AN programming models.

10.11.1 Creating Annotations

Creating an annotation without writing it involves the following steps:

1. Open the HDF file.
2. Initialize the AN interface.
3. Define the type of annotation to be created.
4. Create the annotation.
5. Terminate access to the annotation.
6. Terminate access to the AN interface.
7. Close the file.

To create a file annotation, the calling program must contain the following AN routine calls:

C:

```
file_id = Hopen(filename, access. block_size);
an_id = ANstart(file_id);
ann_id = ANcreatef(an_id, ann_type);
status = ANendaccess(ann_id);
status = ANend(an_id);
status = Hclose(file_id);
```

FORTRAN:

```
file_id = hopen(filename, access, block_size)
an_id = afstart(file_id)
ann_id = affcreate(an_id, ann_type)
status = afendaccess(ann_id)
status = afend(an_id)
status = hclose(file_id)
```

ANstart initializes the multifile annotation interface**. ANcreate** and **ANcreatef** creates the annotation**. ANendaccess** terminates access to the annotation, and **ANend** terminates access to the multifile annotation interface**.**

ANcreate has four parameters: `an_id`, `tag`, `ref` and `ann_type`. The `an_id` parameter is the interface identifier for the annotation to be written, the `tag` and `ref` parameters are the tag/reference number pair of the object the annotation will be assigned to and the `ann_type` parameter is the data annotation type. It is set to either `AN_DATA_LABEL` or `AN_DATA_DESC`. These annotation type definitions are defined in the "hdf.h" header file.

ANcreatef is used to create file annotations. The `ann_type` parameter in a call to **ANcreatef** can only be set to the type definitions `AN_FILE_LABEL` or `AN_FILE_DESC`, which are also defined in the "hdf.h" header file.

ANendaccess's one argument, `ann_id`, is an annotation returned by **ANcreate** and is disposed of by **ANendaccess**. Any subsequent attempts to access the annotation id will result in an error condition.

ANend also has only one parameter, `an_id`, which is the AN interface identifier returned by **ANstart.** This identifier is disposed of by **ANend** and access to the interface routines is terminated during this call. Any subsequent attempts to access the annotation interface id or routines will result in an error condition.

The parameters of these functions are defined in the following table.

TABLE 10K **ANstart, ANcreate, ANcreatef, ANendaccess and ANend Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
ANstart **(afstart)**	file_id	int32	integer	File identifier.
ANcreate **(afcreate)**	an_id	int32	integer	AN interface identifier.
	tag	uint16	integer	Tag of the item to be assigned the data annotation.
	ref	uint16	integer	Reference number of the item to be assigned the data annotation.
	type	int32	integer	Data annotation type: label or description.
ANcreatef **(affcreate)**	an_id	int32	integer	AN interface identifier.
	type	int32	integer	File annotation type: label or description.
ANendaccess **(afendaccess)**	ann_id	int32	integer	Annotation identifier.
ANend **(afend)**	an_id	int32	integer	AN interface identifier.

10.11.2 Writing an Annotation: ANwriteann

The AN programming model for writing an annotation is as follows:

1. Open the HDF file.
2. Initialize the AN interface.
3. Create the annotation identifier.
4. Write the annotation.
5. Terminate access to the annotation.
6. Terminate access to the AN interface.
7. Close the HDF file.

To write a file label, the calling program must contain the following AN routine calls:

C:

```
file_id = Hopen(filename, access. block_size);
an_id = ANstart(file_id);
ann_id = ANcreatef(an_id, ann_type);
status = ANwriteann(ann_id, label, HDstrlen(label));
status = ANendaccess(ann_id);
status = ANend(an_id);
status = Hclose(file_id);
```

FORTRAN:

```
file_id = hopen(filename, access, block_size)
an_id = afstart(file_id)
ann_id = affcreate(an_id, ann_type)
```

```
status = afwriteann(ann_id, label, ann_length)
status = anendaccess(ann_id)
status = afend(an_id)
status = hclose(file_id)
```

ANwriteann has three parameters: `ann_id`, `label` and `ann_length`. The `ann_id` parameter is the identifier for the annotation to be written, the `label` parameter contains the annotation string and the `ann_length` parameter contains the length of the annotation string. The label must be a null-terminated string. In the C interface, the **HDstrlen** function can be used to dynamically determine the length of the label string. In either C or Fortran-77, specifying a length value different from the actual length of the label will result in the label either being truncated or null-padded accordingly.

The parameters of **ANwriteann** are further defined in the following table.

TABLE 10L

ANwriteann Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
ANwriteann (afwriteann)	ann_id	int32	integer	Identifier for the annotation to be written.
	label	char *	character* (*)	Label of the annotation.
	ann_length	int32	integer	Length of the label in bytes.

EXAMPLE 8.

Adding a File Label Using the AN Interface

The following examples add a file label and description to the HDF file named "Example6.hdf" using the AN interface routines.

C:

```
#include "hdf.h"
#include <string.h>

main( )
{

int32 file_id, an_id, ann_id, vgroup_id, ann_type;
uint16 obj_tag, obj_ref;
intn status;
static char file_label[] = "This is a file label.";

/* Create the HDF file. */
file_id = Hopen("Example6.hdf", DFACC_CREATE, 0);

/* Initialize the AN interface and obtain an interface id. */
an_id = ANstart(file_id);

/* Set the file annotation type to be a file label. */
ann_type = AN_FILE_LABEL;

/* Create the file label and obtain an annotation id. */
ann_id = ANcreatef(an_id, ann_type);

/* Write the label to the file. */
status = ANwriteann(ann_id, file_label, strlen(file_label));

/* Create a vgroup. */
status = Vstart(file_id);
vgroup_id = Vattach(file_id, -1, "w");
```

```
status = Vdetach(vgroup_id);
status = Vend(file_id);

obj_tag = DFTAG_VG;
obj_ref = 0;

/* Create the data description label. */
ann_id = ANcreate(an_id, obj_tag, obj_ref, AN_DATA_DESC);

/* Write the label to the file. */
status = ANwriteann(ann_id, file_label, strlen(file_label));

/* Terminate access to the annotation. */
status = ANendaccess(ann_id);

/* Terminate access to the AN interface. */
status = ANend(an_id);

/* Close the file. */
status = Hclose(file_id);

}
```

FORTRAN:

```
      PROGRAM CREATE A FILE LABEL

       integer hopen, afstart, affcreate, afendaccess, afend
       integer hclose, status, file_id, an_id, ann_id
       integer afwriteann

       integer*4 DFACC_CREATE, AN_FILE_LABEL
       parameter (DFACC_CREATE = 4, AN_FILE_LABEL = 2)

       character*21 file_label /"This is a file label."/

C      Open the HDF file.
       file_id = hopen('Example6.hdf', DFACC_CREATE, 0)

C      Initialize the AN interface and obtain an interface id.
       an_id = afstart(file_id)

C      Use the file_id to write the label to the file.
       ann_id = affcreate(an_id, AN_FILE_LABEL)

C      Write the label to the file.
       status = afwriteann(ann_id, file_label, 21)

C      Terminate access to the annotation.
       status = afendaccess(ann_id)

C      Terminate access to the AN interface.
       status = afend(an_id)

C      Close the file.
       status = hclose(file_id)

       end
```

10.12 Reading Annotations Using the AN Interface

As is the case with writing annotations there is only one AN routine used for reading annotations. This routine is described below.

10.12.1 Reading an Annotation: ANreadann

The AN programming model for reading an annotation is similar to the model for writing one:

1. Open the HDF file.
2. Initialize the AN interface.
3. Read the annotation.
4. Terminate access to the annotation.
5. Terminate access to the AN interface.
6. Close the HDF file.

To read a file label, the calling program must contain the following AN routine calls:

C:
```
file_id = Hopen(filename, access. block_size);
an_id = ANstart(file_id);
status = ANreadann(ann_id, buf, HDstrlen(buf));
status = ANendaccess(ann_id);
status = ANend(an_id);
status = Hclose(file_id);
```

FORTRAN:
```
file_id = hopen(filename, access, block_size)
an_id = afstart(file_id)
status = afreadann(ann_id, buf, buf_length)
status = anendaccess(ann_id)
status = afend(an_id)
status = hclose(file_id)
```

ANreadann has three parameters: `ann_id`, `buf` and `buf_length`. The `ann_id` parameter is the identifier for the annotation to be read, the `buf` parameter is the buffer the annotation string will be read into and the `buf_length` parameter is the length of the buffer. As with **ANwriteann**, the **HDstrlen** function can be used to dynamically determine the length of the buffer. In either C or Fortran-77, specifying a buffer length value different from the actual length of the buffer will result in the annotation either being truncated or null-padded accordingly.

The parameters of the **ANreadann** routine is further defined in the following table.

TABLE 10M **ANreadann Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
ANreadann (afreadann)	ann_id	int32	integer	Identifier of the annotation to be read.
	buf	char *	character* (*)	Buffer for the returned annotation.
	buf_length	int32	integer	Size of the buffer in bytes.

10.13 Obtaining Annotation Information Using the AN Interface

In some cases the HDF programmer must get annotation information for the purpose of either locating a particular annotation or set of annotations that correspond to a set of search criteria. The following sections describe the AN routines designed to do this.

10.13.1 Selecting an Annotation: ANselect

The **ANselect** function returns the `ann_id` corresponding to the index passed into the function as the second index parameter and the annotation type passed in as the third `annot_type` parameter. The interface identifier returned by **ANstart** must also be provided. The `annot_type` parameter must be one of the annotation type definitions listed in Section 10.9.2 on page 265. Upon unsuccessful completion **ANselect** returns a `FAIL` status code.

TABLE 10N **ANselect Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
ANselect (afselect)	an_id	int32	integer	AN interface identifier.
	index	int32	integer	Index of the target annotation.
	annot_type	int32	integer	Type of the target annotation.

10.13.2 Obtaining General Annotation Information: ANfileinfo

ANfileinfo returns the total number of file labels, file descriptors, data labels and data descriptors in the current file. The only input parameter required by **ANfileinfo** is the `an_id` passed as its first argument. The information provided by **ANfileinfo** can also be used as object indices. As such they can be passed into a call to **ANselect** within a loop for the purpose of accessing all annotations in a file.

The **ANfileinfo** routine returns either a `SUCCEED` or `FAIL` return code. These status codes are defined in the "hdf.h" header file.

TABLE 10O **ANfileinfo Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
ANfileinfo (affileinfo)	an_id	int32	integer	AN interface identifier.
	n_file_label	int32 *	integer (*)	Number of file labels in the file.
	n_file_desc	int32 *	integer (*)	Number of file descriptors in the file.
	n_data_label	int32 *	integer (*)	Number of data labels in the file.
	n_data_desc	int32 *	integer (*)	Number of data descriptors in the file.

10.13.3 Getting the Length of an Annotation: ANannlen

The **ANannlen** function returns either the length of the annotation corresponding to the `ann_id` passed in the function as it's only argument or a `FAIL` return code upon unsuccessful completion.

TABLE 10P

ANannlen Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
ANannlen (afannlen)	ann_id	int32	integer	Identifier of the target annotation.

EXAMPLE 9.

Reading File Labels

The following examples reads the file label annotation in the HDF file named "Example6.hdf".

C:

```
#include "hdf.h"

main( )
{

int32 file_id, an_id, ann_id, ann_idx, i, ann_length;
int32 n_file_label, n_file_desc, n_data_label, n_data_desc;
int32 status;
char *ann_buf;

/* Open the HDF file for reading. */
file_id = Hopen("Example6.hdf", DFACC_RDWR, 0);

/* Initialize the AN interface and obtain an interface id. */
an_id = ANstart(file_id);

/* Get the annotation information. */
status = ANfileinfo(an_id, &n_file_label, &n_file_desc, \
                &n_data_label, &n_data_desc);

/* Get the file labels. */
for (i = 0; i < n_file_label; i++) {

    /* Get the identifier for the current file label. */
    ann_id = ANselect(an_id, i, AN_FILE_LABEL);

    /* Get the length of the file label. */
    ann_length = ANannlen(ann_id);

    /* Get the length of the file label. */
    ann_buf = HDmalloc(ann_length * sizeof(char));

    /* Read the file label. */
    status = ANreadann(ann_id, ann_buf, ann_length);
}

/* Terminate access to the annotation. */
status = ANendaccess(ann_id);

/* Terminate access to the AN interface. */
status = ANend(an_id);

/* Close the file. */
status = Hclose(file_id);

}
```

FORTRAN:

```
      PROGRAM GET LABELS

       integer*4 afstart, affileinfo, afselect, afendaccess, afend
       integer*4 afannlen, afendaccess, afend, hopen, hclose
       integer*4 status, file_id, an_id, ann_id, i
       character ann_buf(50)

       integer*4 DFACC_RDWR, AN_FILE_LABEL
       parameter (DFACC_RDWR = 3, AN_FILE_LABEL = 2)

C      Open the HDF file.
       file_id = hopen('Example6.hdf', DFACC_RDWR, 0)

C      Initialize the AN interface and obtain an interface id.
       an_id = afstart(file_id)

C      Get the annotation information.
       status = affileinfo(an_id, n_file_label, n_file_desc,
      +                    n_data_label, n_data_desc)

C      Get file label lengths.
       do 10 i = 0, n_file_label

C        Get the identifier for the current index.
         ann_id = afselect(i, AN_FILE_LABEL)

C        Get the length of the annotation for the current id.
         ann_length = afannlen(ann_id)

C        Read the file label.
         status = afreadann(ann_id, ann_buf, ann_length)

10     continue

C      Terminate access to the annotation.
       status = afendaccess(ann_id)

C      Terminate access to the AN interface.
       status = afend(an_id)

C      Close the file.
       status = hclose(file_id)

       end
```

10.13.4 Obtaining the Number of Annotations Corresponding to Given Search Criteria: ANnumann

The **ANnumann** routine returns the total number of annotations in the file that satisfy the search criteria supplied in the latter three parameters of the call. These parameters are: `type`, which is set to one of the type definitions listed in Section 10.9.2 on page 265, `obj_tag`, which is the tag of the object the annotation to be searched for is attached to, and `obj_ref`, which is the reference number of the object the annotation to be searched for is attached to. Upon unsuccessful completion **ANnumann** returns a `FAIL` status code.

TABLE 10Q

ANnumann Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
ANnumann (afnumann)	an_id	int32	integer	AN interface identifier.
	annot_type	int32	integer	Type of the target annotation.
	obj_tag	uint16	integer	Tag of the object the target annotation is attached to.
	obj_ref	uint16	integer	Reference number of the object the target annotation is attached to.

10.13.5 Obtaining the List of Annotations Corresponding to Given Search Criteria: ANannlist

The **ANannlist** routine is the counterpart to **ANnumann** in that, instead of returning the *number* of annotations that satisfy a given set of search criteria supplied in the latter three parameters of the call, it returns the *list* of annotation identifers in the file that satisfy the search criteria. **ANannlist**'s second, third and fourth parameters are the same as **ANnumann**'s. **ANannlist** returns the list in it's fifth parameter. Upon unsuccessful completion **ANannlist** also returns a `FAIL` status code.

TABLE 10R

ANannlist Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
ANannlist (afannlist)	an_id	int32	integer	AN interface identifier.
	annot_type	int32	integer	Type of the target annotation.
	obj_tag	uint16	integer	Tag of the object the target annotation is attached to.
	obj_ref	uint16	integer	Reference number of the object the target annotation is attached to.
	ann_list	int32 *	integer (*)	Buffer for returned annotation identifiers that match the search criteria.

EXAMPLE 10.

Returning the Number of and the List of Object Descriptions

The following examples return the total number and list of object descriptions in the "Example6.hdf" file. The `DFTAG_DIA` tag is defined in the "htags.h" header file as a object description tag. The `AN_DATA_DESC` definition is in the "mfan.h" header file.

C:

```
#include "hdf.h"

main( )
{
int32 status, file_id, an_id, ann_id, ann_num;
int32 *ann_list;
uint16 obj_tag, obj_ref;
int32 annot_type;

/* Create the HDF file. */
file_id = Hopen("Example6.hdf", DFACC_RDWR, 0);

/* Initialize the AN interface and obtain an interface id. */
```

```
an_id = ANstart(file_id);

/* Set the annotation type to be a data description. */
annot_type = AN_DATA_DESC;

/* Set the tag to be a data identifier. */
obj_tag = DFTAG_DIA;

/* Set the reference number to be the first object. */
obj_ref = 0;

/* Get the number of object descriptions. */
ann_num = ANnumann(an_id, annot_type, obj_tag, obj_ref);

/* Allocate space for the annotation identifier. */
ann_list = HDmalloc(ann_num * sizeof(int32));

/* Get the list of object descriptions. */
status = ANannlist(an_id, annot_type, obj_tag, obj_ref, ann_list);

/* Terminate access to the annotation. */
status = ANendaccess(ann_id);

/* Terminate access to the AN interface. */
status = ANend(an_id);

/* Free the space allocated for the annotation identifier. */
HDfree(ann_list);

/* Close the file. */
status = Hclose(file_id);

}
```

FORTRAN:

```
      PROGRAM GET DATA DESC LIST

       integer hopen, afstart, affcreate, afendaccess, afend
       integer hclose status, file_id, an_id, ann_id
       integer annot_type, obj_tag, ann_num
       integer ann_list(50)

       integer*4 DFACC_CREATE, AN_DATA_DESC, DFTAG_DIA
       parameter (DFACC_CREATE = 4, AN_DATA_DESC = 1,
      +           DFTAG_DIA = 105)

C      Open the HDF file.
       file_id = hopen('Example6.hdf', DFACC_RDWR, 0)

C      Initialize the AN interface and obtain an interface id.
       an_id = afstart(file_id)

C      Set the annotation type to be a data description.
       annot_type = AN_DATA_DESC

C      Set the tag to be a data identifier description.
       obj_tag = DFTAG_DIA

C      Set the reference number to be the first object.
       obj_ref = 0

C      Get the number of file labels.
```

```
      ann_num = afnumann(an_id, annot_type, obj_tag, obj_ref)

C     Get the list of file labels.
      status = afannlist(an_id, annot_type, obj_tag, obj_ref, ann_list)

C     Terminate access to the annotation.
      status = afendaccess(ann_id)

C     Terminate access to the AN interface.
      status = afend(an_id)

C     Close the file.
      status = hclose(file_id)

      end
```

10.13.6 Obtaining the Tag/Reference Number Pair From a Specified Annotation Identifiers: ANget_tagref

The **ANget_tagref** routine returns the tag/reference number pair of the annotation identified by the specfied annotation identifier, index and annotation type.

The *ann_index* parameter is zero-based. The *ann_type* parameter value can be either `AN_DATA_LABEL`, `AN_FILE_LABEL`, `AN_DATA_DESC` or `AN_FILE_DESC`.

TABLE 10S **ANget_tagref Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
ANget_tagref (afgettagref)	an_id	int32	integer	AN interface identifier.
	ann_index	int32 *	integer	Index of the target annotation.
	ann_type	int32 *	integer	Annotation type of the target annotation.
	ann_tag	uint16 *	integer	Tag of the target annotation.
	ann_ref	uint16 *	integer	Reference number of the target annotation.

10.13.7 Obtaining the Tag/Reference Number Pair From a Specified Annotation Identifier: ANid2tagref

The **ANid2tagref** routine returns the tag/reference number pair of the annotation that is also identified by the specified annotation id.

ANid2tagref returns the tag/reference number pair if successful and `FAIL` (or `-1`) otherwise.

TABLE 10T **ANid2tagref Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
ANid2tagref (afidtagref)	an_id	int32	integer	AN interface identifier.
	ann_tag	uint16 *	integer	Tag of the target annotation.
	ann_ref	uint16 *	integer	Reference number of the target annotation.

10.13.8 Obtaining the Annotation Identifier From a Specified Tag/ Reference Number Pair: ANtagref2id

The **ANtagref2id** routine returns the identifier of the annotation that is also identified by the specified tag/reference number pair.

ANtagref2id returns the annotation identifier if successful and `FAIL` (or `-1`) otherwise.

TABLE 10U **ANtagref2id Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type C	Data Type Fortran-77	Description
ANtagref2id (aftagrefid)	an_id	int32	integer	AN interface identifier.
	ann_tag	uint16 *	integer	Type of the target annotation.
	ann_ref	uint16	integer	Tag of the object the target annotation is attached to.

10.13.9 Obtaining an Object Tag From a Specified Annotation Type: ANatype2tag

The **ANatype2tag** routine returns the object tag (prefaced by `DFTAG_`) corresponding to a given annotation type (prefaced by `AN_`).

Return values are `DFTAG_FID`, `DFTAG_FD`, `DFTAG_DIL or DFTAG_DIA`.

TABLE 10V **ANatype2tag Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type C	Data Type Fortran-77	Description
ANatype2tag	tag	int32 *	integer	Object type.

10.13.10 Obtaining an Annotation Type From a Specified Object Tag: ANtag2atype

The **ANtag2atype** routine returns theannotation type (prefaced by `AN_`) corresponding to a given object tag (prefaced by `DFTAG_`).

Return values are `AN_FILE_LABEL`, `AN_FILE_DESC`, `AN_DATA_LABEL`, `AN_DATA_DESC`.

TABLE 10W **ANtag2atype Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type C	Data Type Fortran-77	Description
ANtag2atype (aftagatype)	ann_type	uint16	integer	Annotation type.

Chapter 11

Scientific Data Sets (DFSD API)

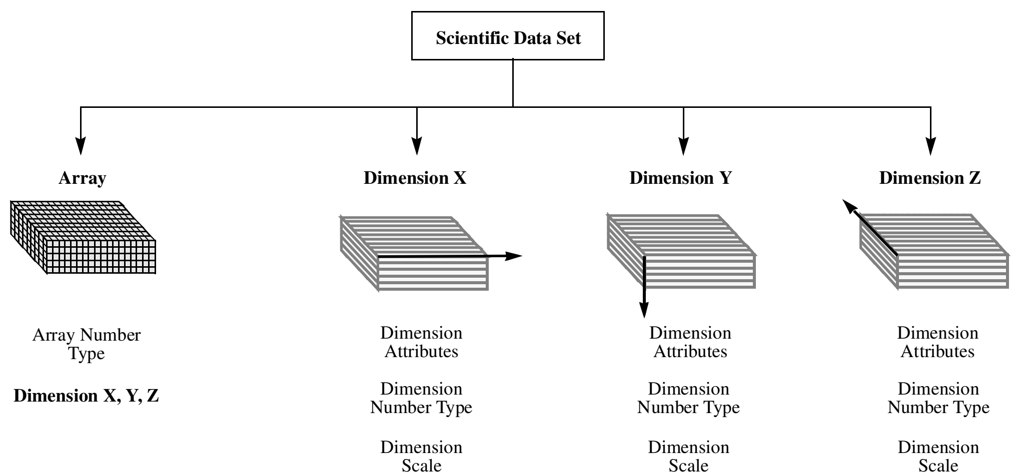
11.1 Chapter Overview

The DFSD interface is one of two interfaces in the HDF library that support the scientific data model. With the release of HDF version 3.3, the multifile SD interface described in Chapter 3, titled *Scientific Data Sets (SD API)*, was made available. Generally, the SD interface should be used instead of the DFSD interface.

11.2 The DFSD Scientific Data Set Data Model

The scientific data set, or SDS, data model supports four primary data objects: arrays, dimensions, dimension scales, and dimension attributes. As in the multifile SD SDS model, the fundamental object of the data model is the SDS array. Unlike the SD multifile SDS model the DFSD SDS model has, in addition to dimension attributes, attributes that refer to the SDS array itself.

FIGURE 11a **The Contents of a Three-Dimensional DFSD Scientific Data Set**



11.2.1 Required DFSD SDS Objects

The only required objects in the DFSD SDS model are the ***array*** and the ***number type*** of the array data. Without this information, the data set is inaccessible. Required objects are created by the library using the information supplied at the time the SDS is defined.

Descriptions of these objects in Chapter 3, titled *Scientific Data Sets (SD API)*.

11.2.1.1 Dimensions

Unlimited dimensions, supported in the multifile SD SDS model, aren't supported in the single-file DFSD SDS model.

11.2.2 Optional DFSD SDS Objects

There are two types of optional objects available for inclusion in an SDS: dimension scales and attributes. Optional objects are only created when specified by the calling program.

11.2.2.1 Dimension Scales

Conceptually, a***dimension scale*** is a series of numbers placed along a dimension to demarcate intervals in a data set. They are assigned one per dimension. Structurally, each dimension scale is a one-dimensional array with size and name equal to its assigned dimension name and size.

11.2.2.2 Predefined Attributes

Predefined attributes are attributes that have reserved labels and in some cases predefined number types. They are described in Chapter 3, titled *Scientific Data Sets (SD API)*.

11.3 The Single-File Scientific Data Set Interface

The HDF library currently contains several routines for storing scientific data sets in the HDF format. **DFSDadddata**, **DFSDputdata**, and **DFSDgetdata** perform data I/O and by default assume that all scientific data is uncompressed 32-bit floating-point data stored in row-major order. DFSD library routines also read and write subsets and slabs of data, set defaults, determine the number of data sets in a file, and inquire about or assigning reference numbers before reading or writing data.

11.3.1 DFSD Library Routines

The names of the C routines in the DFSD library are prefaced by "DFSD" and the names of the equivalent Fortran-77 functions are prefaced by "ds". They are categorized as follows:

- ***Write routines*** create new data sets and add slabs to existing data sets.
- ***Read routines*** read whole scientific data sets.
- ***Slab routines*** read and write subsets and slabs of scientific data.
- ***Data set attribute routines*** read and write the predefined string and value attributes assigned to data sets.
- ***Dimension attribute routines*** read and write the predefined string and value attributes assigned to dimensions.

DFSD library routines are more explicitly defined in Table 11A and on their respective pages in the *HDF Reference Guide*.

TABLE 11A **DFSD Library Routines**

Category	Routine Name		Description
	C	Fortran-77	
Write	DFSDadddata	dsadata	Appends a data set to a file.
	DFSDclear	dsclear	Clears all possible set values.
	DFSDputdata	dspdata	Overwrites new data to a file.
	DFSDsetdims	dssdims	Sets the rank and dimension for succeeding data sets.
	DFSDsetNT	dssnt	Sets the number type for the data set.
	DFSDwriteref	dswref	Assigns a reference number to the next data set written.
Read	DFSDgetdata	dsgdata	Retrieves the next data set in the file.
	DFSDgetdims	dsgdims	Returns the number and dimensions for the next data set.
	DFSDgetNT	dsgnt	Determines the number type for the data in the data set.
	DFSDlastref	dslref	Returns the reference number of last data set accessed.
	DFSDndatasets	dsnum	Returns the number of data sets in a file.
	DFSDpre32sdg	dsp32sd	Determines if the data set was created before HDF version 3.2.
	DFSDreadref	dsrref	Locates a data set with the specified reference number.
	DFSDrestart	dsfirst	Sets the location of the next access operation to be the first data set in the file.
Slabs	DFSDendslab	dssslab	Terminates a read or write slab operation.
	DFSDreadslab	dsrslab	Reads a slab of data from a data set.
	DFSDstartslab	dssslab	Begins a read or write slab operation.
	DFSDwriteslab	dswslab	Writes a slab of data to a data set.
Data Set Attribute	DFSDgetcal	dsgcal	Retrieves the calibration information for the data se.t
	DFSDgetdatalen	dsgdaln	Retrieves the length of the attributes assigned to the data.
	DFSDgetdatastrs	dsgdast	Returns the label, unit, format and coordinate system for data.
	DFSDgetfillvalue	dsgfill	Retrieves the fill value used to complete the data set.
	DFSDgetrange	dsgrang	Retrieves the range of values for the data set.
	DFSDsetcal	dsscal	Sets the calibration information for the data set.
	DFSDsetdatastrs	dssdast	Sets label, unit, format and coordinate system for data.
	DFSDsetfillvalue	dssfill	Sets the fill value to use when completing a data set.
	DFSDsetlengths	dsslens	Sets the length for the data set and dimension attributes.
	DFSDsetrange	dssrang	Sets the range of values for the data set.
Dimension Attribute	DFSDgetdimlen	dsgdiln	Retrieves the length of the attributes assigned to the dimension.
	DFSDgetdimscale	dsgdisc	Returns the scale for a dimension.
	DFSDgetdimstrs	dsgdist	Returns the label, unit, and format for a dimension.
	DFSDsetdimscale	dssdisc	Sets the scale for a dimension.
	DFSDsetdimstrs	dssdist	Sets the label, unit and format for the dimension.

11.3.2 File Identifiers in the DFSD Interface

File identifiers are handled internally by each routine and access to a file is granted simply by providing a filename. As the file identifier is handled by the function call, the calling program need not keep track of how to open and close files.

11.4 Writing DFSD Scientific Data Sets

The DFSD programming model for writing an SDS to an HDF file involves the following steps:

1. Define data set options. (optional)

2. Write all or part of the data set.

These steps are performed for every data set written to a file. However, it is not always necessary to define data set options for every write operation as setting an option places information about the data set in a structure in primary memory. This information is retained until explicitly altered by another set call.

11.4.1 Creating a DFSD Scientific Data Set: DFSDadddata and DFSDputdata

To define and write a single SDS, the calling program must contain of of the following routines:

C: `status = DFSDadddata(filename, rank, dim_sizes, data);`

FORTRAN: `status = dsadata(filename, rank, dim_sizes, data)`

OR

C: `status = DFSDputdata(filename, rank, dim_sizes, data);`

FORTRAN: `status = dspdata(filename, rank, dim_sizes, data)`

DFSDadddata appends data to a file when given an existing file name and creates a new file when given a unique file name. **DFSDputdata** replaces the contents of a file when given an existing file name and creates a new file when given a unique file name. To avoid accidentally overwriting data in a file, the use of **DFSDadddata** is recommended.

DFSDadddata and **DFSDputdata** have four parameters: `filename`, `rank`, `dim_sizes`, and `data`. In both routines, the data set is written to the file specified by the `filename` parameter. The total number of dimensions in the array and the size of each dimension are passed in the `rank` and `dim_sizes` parameters. A pointer to the data or slab of data written to the named file is passed in the `data` parameter.

The parameters of **DFSDadddata** and **DFSDputdata** are further described in the following table.

TABLE 11B **DFSDadddata and DFSDputdata Parameter List.**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFSDadddata (dsadata)	filename	char *	character* (*)	Name of the file containing the data set.
	rank	int32	integer	Number of dimensions in the array.
	dim_sizes	int32 *	integer (*)	Size of each dimension in the data array.
	data	VOIDP	<valid numeric data type>	Array containing the data.
DFSDputdata (dsadata)	filename	char *	character* (*)	Name of file containing the data set.
	rank	int32	integer	Number of dimensions in the array.
	dim_sizes	int32 *	integer (*)	Size of each dimension in the data array.
	data	VOIDP	<valid numeric data type>	Array containing the data.

11.4.2 Specifying the Data Type of a DFSD SDS: DFSDsetNT

The default data type for scientific data is `DFNT_FLOAT32`. To change the default setting, the calling program must contain calls to the following routines:

C:
```
status = DFSDsetNT(number_type);
status = DFSDadddata(filename, rank, dim_sizes, data);
```

FORTRAN:
```
status = dssnt(number_type)
status = dsadata(filename, rank, dim_sizes, data)
```

DFSDsetNT defines the data type for all subsequent **DFSDadddata** and **DFSDputdata** calls until it is changed by a subsequent call to **DFSDsetNT** or reset to the default by **DFSDclear**. **DFSDsetNT**'s only parameter is the data type.

EXAMPLE 1.

Creating and Writing to a DFSD Scientific Data Set

In the following code examples, **DFSDadddata** is used to write an array of 64-bit floating-point numbers to a file named "Example1.hdf". Although the **DFSDsetNT** function call is optional, it is included here to demonstrate how to override the `float32` default.

C:

```
#include "hdf.h"

#define LENGTH 3
#define HEIGHT 2
#define WIDTH 5

main( )
{

      /* Create data array - store dimensions in array 'dims' */
      static float64 scien_data[LENGTH][HEIGHT][WIDTH] =
        { 1., 2., 3., 4., 5.,
        6., 7., 8., 9.,10.,
        11.,12.,13.,14.,15.,
        16.,17.,18.,19.,20.,
        21.,22.,23.,24.,25.,
        26.,27.,28.,29.,30. };

      intn status;

      int32 dims[3] = {LENGTH, HEIGHT, WIDTH};

      /* Set number type to 64-bit float */
      status = DFSDsetNT(DFNT_FLOAT64);

      /* Write the data to file */
      status = DFSDadddata("Example1.hdf", 3, dims, scien_data);

}
```

FORTRAN:

```
       PROGRAM WRITE SDS

      integer  dsadata, dssnt, dims(3), status
      real*8   sci_data(5,2,3)

C     Create array called 'sci_data'; store dimensions in array 'dims'.
      data     sci_data/ 1., 2., 3., 4., 5.,
     $                   6., 7., 8., 9.,10.,
     $                   11.,12.,13.,14.,15.,
     $                   16.,17.,18.,19.,20.,
     $                   21.,22.,23.,24.,25.,
     $                   26.,27.,28.,29.,30./

      data dims /3,2,5/
```

```
C     Set number type to 64-bit float
      status = dssnt(6)

C     Write the data to file
      status = dsadata('Example1.hdf', 3, dims, sci_data)

      end
```

11.4.3 Overwriting Data for a Given Reference Number: DFSDwriteref

DFSDwriteref is a highly specialized function call that overwrites data referred to by the specified reference number.

If **DFSDwriteref** is called with a reference number that doesn't exist, an error return value of -1 will be returned.

The following series of function calls should appear in your program:

C:

```
status = DFSDwriteref(filename, ref_number);
status = DFSDadddata(filename, rank, dim_sizes, data);
```

FORTRAN:

```
status = dswref(filename, ref_number)
status = dsadata(filename, rank, dim_sizes, data)
```

If the filename passed to **DFSDwriteref** is different from the filename in the **DFSDadddata** or **DFSDputdata** routine calls, it will be ignored. The next scientific data set written, regardless of the filename, is assigned the reference number `ref_number`.

Care should be taken when using **DFSDwriteref**, as once the new data has been written the old data cannot be retrieved.

The parameters of **DFSDwriteref** are described in the following table.

TABLE 11C **DFSDsetNT and DFSDwriteref Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFSDsetNT (dssNT)	number_type	int32	integer	Number type tag.
DFSDwriteref (dswref)	filename	char *	character* (*)	Name of the file containing the data.
	ref_number	int16	integer	Reference number to be assigned to the data setcreated.

11.4.4 Writing Several Data Sets: DFSDsetdims and DFSDclear

The DFSD programming model for writing multiple data sets to an HDF file is identical to that for writing individual data sets. (Refer to Section 11.4 on page 281). To understand how multiple data sets are written to file, it is first necessary to take a closer look at each step of the programming model. First and most importantly, all DFSD routines that set a write option except **DFSDsetNT** and **DFSDsetfillvalue** add information to a special structure in primary memory. This information is used to determine how data is written to file for all subsequent write operations.

Information stored in primary memory is retained by the HDF library until explicitly changed by a call to **DFSDsetdims** or reset to `NULL` by calling **DFSDclear**. **DFSDsetdims** and **DFSDclear** are used to prevent assignments of attributes created for a group of data sets to data sets outside the group. For more information on assigning attributes see Section 11.7.1 on page 293 and Section 11.7.3 on page 299.

11.4.5 Preventing the Reassignment of DFSD Data Set Attributes: DFSDsetdims

Information stored in primary memory is retained by the HDF library until explicitly changed by a call to **DFSDsetdims** or reset to `NULL` by calling **DFSDclear**. **DFSDsetdims** and **DFSDclear** are used to prevent assignments of attributes created for a group of data sets to data sets outside the group.

The syntax of **DFSDsetdims** is the following:

C:
```
status = DFSDsetdims(rank, dim_sizes);
```

FORTRAN:
```
status = dssdims(rank, dim_sizes)
```

DFSDsetdims is not used here to define the rank and dimension sizes to be used in the next operation, but to alert the DFSD interface to stop the automatic assignment of attributes to the data sets to be written to file. **DFSDsetdims** has two parameters: `rank` and `dim_sizes`. The rank of an array is the total number of dimensions in the array and the dimension sizes are the length of each individual dimension.

As a rule of thumb, **DFSDsetdims** should be called if any **DFSDset*** routine (**DFSDsetNT**, for example) has been called. This insures that all attribute values that have been reset will be assigned in future data set operations.

The parameters of **DFSDsetdims** are further defined in the following table.

TABLE 11D **DFSDsetdims Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFSDsetdims **(dssdims)**	rank	intn	integer	Number of dimensions in the array.
	dim_sizes	int32*	integer (*)	Size of each dimension in the array.

11.4.6 Resetting the Default DFSD Interface Settings: DFSDclear

The syntax for **DFSDclear** is as follows:

C:
```
status = DFSDclear( );
```

FORTRAN:
```
status = dsclear( )
```

The **DFSDclear** routine clears all interface settings defined by any of the **DFSDset** routines (**DFSDsetNT**, **DFSDsetfillvalue**, **DFSDsetdims**, **DFSDsetdatastrs**, **DFSDsetdatalengths**, **DFSDsetrange**, **DFSDsetcal**, **DFSDsetdimscale** and **DFSDsetdimstrs**). After the **DFSDclear** has been called, calls to any of the **DFSDset** routines will result in the corresponding value not being written. To write new values, call the appropriate **DFSDset** routine again.

TABLE 11E **DFSDclear Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFSDclear (**dsclear**)	None	None	None	Clears all DFSD interface settings.

11.5 Reading DFSD Scientific Data Sets

The DFSD programming model for reading an SDS is also a two-step operation:

1. Obtain information about the data set if necessary.
2. Read all or part of the data set.

These steps are performed for every data set read. In some cases, calls to determine the data set definition may be reduced or avoided completely. For example, if the data set dimensions are known, the call that returns the data set dimensions may be eliminated.

11.5.1 Reading a DFSD SDS: DFSDgetdata

If the dimensions of the data set are known**, DFSDgetdata** is the only function call required to read an SDS. If the file is being opened for the first time, **DFSDgetdata** returns the first data set in the file. Any subsequent calls will return successive data sets in the file - data sets are read in the same order they were written. Normally, **DFSDgetdims** is called before **DFSDgetdata** so that space allocations for the array can be checked if necessary and the dimensions verified. If this information is already known, **DFSDgetdims** may be omitted.

To read an SDS of known dimension and number type, the calling program should include the following routine:

C: `status = DFSDgetdata(filename, rank, dim_sizes, data);`

FORTRAN: `status = dsgdata(filename, rank, dim_sizes, data)`

DFSDgetdata has four parameters: `filename`, `rank`, `dim_sizes`, and `data`. **DFSDgetdata** returns a data set specified by the parameter `filename`. The total number of dimensions is specified in `rank` and the size of each dimension is specified in `dim_sizes`. **DFSDgetdata** returns the array in `data`.

The parameters of **DFSDgetdata** are further defined in the following table.

TABLE 11F **DFSDgetdata Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	

DFSDgetdata (dsgdata)	filename	char	character* (*)	Name of the file containing the data.
	rank	int32	integer	Number of dimensions.
	dim_sizes	int32 *	integer (*)	Buffer for the dimension sizes.
	data	VOIDP	<valid numeric data type>	Buffer for the stored scientific dat.

11.5.2 Specifying the Dimensions and Data Type of an SDS: DFSDgetdims and DFSDgetNT

When **DFSDgetdims** is first called, it returns dimension information of the first data set. Subsequent calls will return this information for successive data sets. If you need to determine the dimensions or the data type of an array before reading it, call **DFSDgetdims** and **DFSDgetNT**. **DFSDgetNT** gets the data type (or, in HDF parlance, number type) of the data retrieved in the next read operation.

To determine the dimensions and data type of an array before attempting to read it, the calling program must include the following:

C:
```
status = DFSDgetdims(filename, rank, dimsizes, max_rank);
status = DFSDgetNT(number_type);
status = DFSDgetdata(filename, rank, dimsizes, data);
```

FORTRAN:
```
status = dsgnt(filename, rank, dimsizes, max_rank)
status = dsgdims(number_type)
status = dsgdata(filename, rank, dimsizes, data)
```

DFSDgetdims has four parameters: `filename`, `rank`, `dim_sizes`, and `maxrank`. The number of dimensions is returned in `rank`, the size of each dimension in the array `dim_sizes`, and the size of the array containing the dimensions sizes in `max_rank`. **DFSDgetNT** has only one parameter: `number_type`. As there is no way to specify the file or data set through the use of **DFSDgetNT**, it is only valid if it is called after **DFSDgetdims**.

The parameters of **DFSDgetdims** and **DFSDgetNT** are further defined in the following table.

TABLE 11G **DFSDgetNT and DFSDgetdims Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFSDgetdims (dsgdims)	filename	char *	character* (*)	Name of file containing the data.
	rank	intn *	integer	Number of dimensions.
	dim_sizes	int32 *	integer	Buffer for the dimension sizes.
	max_rank	intn	integer	Size of the dimension size buffer.
DFSDgetNT (dsgnt)	number_type	int32 *	integer	Data type of the data to be read.

EXAMPLE 2. **Reading from a DFSD Scientific Data Set**

The following examples search the file named "Example1.hdf" for the dimensions and data type of a DFSD array. Although use of **DFSDgetdims** and **DFSDgetNT** is optional, they are included here as a demonstration of how to verify the array dimensions and number type before reading any data. If the dimensions and type are known, only a call to **DFSDgetdata** is required.

C:

```
#include "hdf.h"

#define LENGTH 3
#define HEIGHT 2
#define WIDTH 5

main( )
{

      float64 scien_data[LENGTH][HEIGHT][WIDTH];
      int32 number_type;
      intn rank, status;
      int32 dims[3];

      /* Get the dimensions and number type of the array */
      status = DFSDgetdims("Example1.hdf", &rank, dims, 3);
      status = DFSDgetNT(&number_type);

      /* Read the array if the dimensions are correct */
      if (dims[0] <= LENGTH && dims[1] <= HEIGHT && dims[2] <= WIDTH)
        status = DFSDgetdata("Example1.hdf", rank, dims, scien_data);

}
```

FORTRAN:

```
       PROGRAM READ SDS

      integer  dsgdata, dsgdims, dsgnt, dims(3), status
      integer rank, num_type
      real*8   sci_data(5, 2, 3)

C     Get the dimensions and number type of the array.
      status = dsgdims('Example1.hdf', rank, dims, 3)
      status = dsgnt(num_type)

C     Read the array if the dimensions are correct.
      if ((dims(1) .eq. 3) .and. (dims(2) .eq. 2) .and.
     +    (dims(3) .eq. 5)) then
            status = dsgdata('Example1.hdf', rank, dims, sci_data)
      endif

      end
```

11.5.3 Determining the Number of DFSD Data Sets: DFSDndatasets and DFSDrestart

DFSDgetdims and **DFSDgetdata** sequentially access DFSD data sets. By repeatedly calling either function, a program can step through an entire file by reading one data set at a time. However, before attempting to sequentially access all of the data sets in a file the total number of data sets in the file should be determined. To do so, the calling program must call the following routine:

C: `num_of_datasets = DFSDndatasets(filename);`

FORTRAN: `num_of_datasets = dsnum(filename)`

Once the total number of data sets is known, a calling program can at any time, reset the current data set to the first data set in the file by calling the following routine:

C:
```
status = DFSDrestart( );
```

FORTRAN:
```
status = dsfirst( )
```

Use of **DFSDndatasets** and **DFSDrestart** is optional, it is usually more convenient than cycling through the entire file one SDS at a time.

11.5.4 Obtaining Reference Numbers of DFSD Data Sets: DFSDreadref and DFSDlastref

As the HDF library handles the assignment and tracking of reference numbers, reference numbers must be explicitly returned. Obtaining the reference number is an operation best performed immediately after data set creation.

The DFSD interface uses the function **DFSDreadref** to initiate access to individual scientific data sets. **DFSDreadref** specifies the reference number of the next SDS to be read.

To access a specific SDS, the calling program must contain the following routines:

C:
```
status = DFSDreadref(filename, ref);
status = DFSDgetdata(filename, rank, dim_sizes, data);
```

FORTRAN:
```
status = dsrref(filename, ref)
status = dsgdata(filename, rank, dim_sizes, data)
```

DFSDreadref has two parameters: `filename` and `ref`. **DFSDreadref** specifies the reference number of the object to be next operated on in the HDF file `filename` as `ref`. Determining the correct reference number is the most difficult part of this operation. As a result, **DFSDreadref** is often used in conjunction with **DFSDlastref**, which determines the reference number of the last data set accessed.

To syntax of **DFSDadddata** and **DFSDlastref** is:

C:
```
status = DFSDadddata(filename, rank, dim_sizes, data);
ref_num = DFSDlastref( );
```

FORTRAN:
```
status = dsadata(filename, rank, dim_sizes, data)
ref_num = dslref( )
```

DFSDputdata can also be used with **DFSDlastref** to obtain similar results. In any case, **DFSDlastref** can be used before any operation that requires identifying a scientific data set by reference number, as in the assignment of annotations and inserting data sets into vgroups. For more information about annotations and vgroups refer to Chapter 10, titled *Annotations (DFAN API)*, and Chapter 5, titled *Vgroups (V API)*.

TABLE 11H **DFSDreadref Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFSDreadref **(dsrref)**	filename	char *	character* (*)	Name of the file containing the data set.
	ref_number	uint16	integer	Reference number of the next data set to be read.

11.6 Slabs in the DFSD Interface

To review, a***slab*** is an n-dimensional array whose dimensions are smaller than those of the SDS array into which it is written or from which it is read.

11.6.1 Accessing Slabs: DFSDstartslab and DFSDendslab

There are two routines required for every DFSD slab operation - **DFSDstartslab** and **DFSDendslab**. **DFSDstartslab** is used to initialize the slab interface and to initiate access to new or existing data sets. **DFSDendslab** is used to terminate access to open data sets. **DFSDstartslab** must be called before any read or write slab operation and **DFSDendslab** must be called after the slab operation is completed. Both routines are required when reading and writing slabs.

Given a new filename, **DFSDstartslab** will create a new HDF file with the specified name. Given an existing filename, it will open the named file and append the new data set to the end of the file. Its only parameter is `filename`. **DFSDendslab** has no parameters and need only be called once per file. **DFSDendslab** will write any attributes defined immediately before the data set is created.

For more information on assigning attributes, see Section 11.7.3 on page 299.

TABLE 11I **DFSDstartslab Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFSDstartslab (dssslab)	filename	char *	character* (*)	Name of the file containing the data set.

11.6.2 Writing Slabs: DFSDwriteslab

In the DFSD interface, writing an entire data set array and writing slabs follow the same programming model. The difference between the two is that calls to three routines is needed to write slabs, while a call to one routine is needed to write whole data sets.

More specifically, the DFSD programming model for writing slabs to an SDS is as follows:

1. Set the appropriate options to define the new SDS or select an existing SDS.
2. Write the data set using three specialized slab routines.

In addition to writing slabs to both new and existing data sets, **DFSDwriteslab** can also perform the following sequential write operations:

- Write slabs to a single data set when called repeatedly.
- Write slabs to sequential data sets when repeatedly called between calls to **DFSDgetdims**.
- Write slabs to selected data sets when repeatedly called between calls to **DFSDwriteref**.

Although not specifically defined as a slab routine, in practice, the **DFSDsetfillvalue** routine is used to initialize array elements between non-contiguous slab write operations. Setting a fill value places the same value in every array location before the first slab is written. Any hole created by non-contiguous writes can then be recognized by identifying the known fill value. The fill value must have the same number type as the values in the data set. For more information on fill values refer to Section 11.7.1.2 on page 294.

To write a slab to a new data set, the calling program must include the following routine calls:

C:
```
status = DFSDsetdims(rank, dimsizes);
```

```
status = DFSDsetNT(num_type);
status = DFSDstartslab(filename);
status = DFSDwriteslab(start, stride, count, data);
status = DFSDendslab( );
```

FORTRAN:
```
status = dssnt(num_type)
status = dssdims(rank, dim_sizes)
status = dssslab(filename)
status = dswslab(start, stride, edge, data)
status = dseslab( )
```

When writing slabs to an existing data set, it is impossible to change the number type, array boundaries, fill value, or calibration information. Consequently **DFSDsetNT**, **DFSDsetdims**, **DFSDsetcal**, and **DFSDsetfillvalue** will generate errors if called for an existing data set.

To write a slab to an existing data set, your program should include the following calls:

C:
```
status = DFSDwriteref(filename, ref);
status = DFSDstartslab(filename);
status = DFSDwriteslab(start, stride, count, data);
status = DFSDendslab( );
```

FORTRAN:
```
status = dswref(filename, ref)
status = dssslab(filename)
status = dswslab(start, stride, edge, data)
status = dseslab( )
```

Because **DFSDwriteslab** offers no overwrite protection, the calling program is responsible for eliminating overlap when arranging slabs within the newly defined data set.

DFSDwriteslab has four arguments: `start`, `stride`, `edge`, and `data`. The arguments `start`, `stride`, and `edge` are defined as they are in the corresponding SD routines.

The DFSD SDS model does not support strides. Pass the `start` array as the `stride` parameter as a place holder. Whatever is passed as the `stride` parameter will be ignored by the DFSD interface.

Although **DFSDendslab** need only be called once per file, it is required to write data to the file. It will also write any attributes defined immediately before the data set is created.

TABLE 11J **DFSDwriteslab Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFSDwriteslab (sdwslab)	start	int32 *	integer (*)	Array containing the starting coordinate the write.
	stride	int32 *	integer (*)	Ignored parameter.
	count	int32 *	integer (*)	Array defining the boundaries of the slab.
	data	VOIDP	<valid numeric data type>	Buffer for the data to be written.

11.6.3 Reading Slabs: DFSDreadslab

The programming model for reading one or more slabs involves the following steps:

1. Select an existing SDS.
2. Read the data set using three specialized slab routines.

In addition to reading single slabs of data, **DFSDreadslab** can perform the following sequential access operations:

- Read multiple slabs from the first data set in a file when called repeatedly.
- Read multiple slabs from a specified data set when repeatedly called after **DFSDreadref**.
- Read multiple slabs from sequential data sets when repeatedly called between calls to **DFSDgetdims**.

To read a slab, the calling program must include the following routine calls:

C:
```
status = DFSDreadref(filename, ref);
status = DFSDstartslab(filename);
status = DFSDreadslab(start, stride, edge, data);
status = DFSDendslab( );
```

FORTRAN:
```
status = dsrref(filename, ref)
status = dssslab(filename)
status = dsrslab(start, stride, edge, data)
status = dseslab( )
```

In addition to **DFSDreadref**, **DFSDgetdims** may also be used to position the read pointer to the appropriate data set. When **DFSDreadslab** is used to read slabs, the coordinates of the `start` array must begin at 0 for each dimension (`start={0,0, ... 0}`) and the size of each dimension must equal the size of the array itself (`edge={dim_size_1, dim_size_2, dim_size_n}`). As with **DFSDwriteslab**, whatever is passed in as the `stride` parameter is ignored. Finally, the `data` buffer must allocate enough space to hold the data: excess data is truncated.

All parameters of the **DFSDreadslab** routine assume Fortran-77-style one-based arrays - the starting coordinates of the slab must be given as an offset from the origin of the data set where the origin is defined as `(dim 1 = 1, dim 2 = 1, . . . dim n-1 = 1, dim n = 1)`. The first element of the slab will be the coordinates specified by the contents of the `start` array. **DFSDreadslab** will extract elements in increasing order until the until the dimensional offset specified by the contents of the `edge` array are encountered.

TABLE 11K **DFSDreadslab Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFSDreadslab (dswslab)	filename	char *	character* (*)	Name of the HDF file.
	start	int32 *	integer (*)	Array containing the coordinates for start of the slab.
	slab_size	int32	integer (*)	Array of rank containing the size of each dimension of the slab.
	stride	int32 *	integer (*)	Place holder array.
	buffer	VOIDP	<valid numeric data type>	Array the will used to store the extracted slab.
	buffer_size	int32 *	integer (*)	Array containing the dimensions of the `buffer` parameter.

11.7 Predefined Attributes and the DFSD Interface

Although they often contain important information, attributes are optional to the data set array and the dimension record. Although both types of attributes use similar names, they are read and written using different sets of routines. All attributes are predefined by the DFSD library.

11.7.1 Writing Data Set Attributes

Data set attributes are described in Chapter 3, titled *Scientific Data Sets (SD API)*. There is a limit of one string attribute per data set.

11.7.1.1 Assigning String Attributes to an SDS: DFSDsetlengths and DFSDsetdatastrs

The DFSD interface provides two function calls for creating data set string attributes: **DFSDsetlengths** and **DFSDsetdatastrs**. **DFSDsetlengths** overrides the default string length and **DFSDsetdatastrs** writes the string. **DFSDsetlengths** and **DFSDsetdatastrs** are optional and may be called individually, or in any order as long as they precede calls to **DFSDadddata** or **DFSDputdata**.

Predefined string attributes are defined as follows:

- ***Coordinate system attributes*** specify the coordinate system used to generate the original data.
- ***Format attributes*** specify the format to use when displaying values for the data.
- ***Label attributes*** contains data array names.
- ***Unit attribute*s** identifies the units of measurement associated with the data.

To assign a predefined attribute to an HDF file, the program must contain the following routine calls:

C:
```
status = DFSDsetlengths(label_len, unit_len, format_len,
                        coords_len);
status = DFSDsetdatastrs(label, unit, format, coordsys);
status = DFSDadddata(filename, rank, dimsizes, data);
```

FORTRAN:
```
status = dsslens(label_len, unit_len, format_len, coords_len)
status = dssdast(label, unit, format, coordsys)
status = dsadata(filename, rank, dimsizes, data)
```

DFSDsetlengths has four arguments: `label_len`, `unit_len`, `format_len`, and `coords_len`. Each parameter reflects the maximum length for the string that will hold the label, unit, format, and coordinate system. Use of **DFSDsetlengths** is optional and usually not necessary.

DFSDsetdatastrs writes null-terminated strings to an HDF file. It has the same four arguments: `label`, `unit`, `format`, and `coordsys`. To avoid the assignment of a string, pass `NULL` as the appropriate argument.

TABLE 11L **DFSDsetlengths and DFSDsetdatastrs Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFSDsetlengths (dsslens)	label_len	intn	integer	Maximum length for any label string.
	unit_len	intn	integer	Maximum length for any unit string.
	format_len	intn	integer	Maximum length for any format string.
	coords_len	intn	integer	Maximum length for any coordinate system string.
DFSDsetdatastrs (dssdast)	label	char *	character* (*)	Label describing the data.
	unit	char *	character* (*)	Unit to be applied to the data.
	format	char *	character* (*)	Format to be applied in displaying the data.
	coordsys	char*	character* (*)	Coordinate system of the data set.

11.7.1.2 Assigning Value Attributes to a DFSD SDS: DFSDsetfillvalue, DFSDsetrange, DFSDsetcal

The DFSD interface provides the following routines for defining value attributes. All three function calls are optional and may be called in any order provided they precede a call to **DFSDadddata** or **DFSDputdata**.

To assign a value attribute to a data set, the following routines must be called:

C:
```
status = DFSDsetfillvalue(fill_val);
status = DFSDsetcal(scale, scale_err, offset, offset_err,
                    num_type);
status = DFSDsetrange(max, min);
status = DFSDadddata(filename, rank, dimsizes, data);
```

FORTRAN:
```
status = dssfill(fill_val)
status = dsscal(scale, scale_err, offset, offset_err, num_type)
status = dssrang(max, min)
status = dsadata(filename, rank, dimsizes, data)
```

DFSDsetrange sets a new range attribute for the current DFSD SDS**. DFSDsetrange** has two arguments: `max` and `min`. The HDF library will not check or update the range attributes as new data are added to the file, therefore `max` and `min` will always reflect the values supplied by the last **DFSDsetrange** call. The parameters for **DFSDsetrange** is defined in Table 11K below.

DFSDsetfillvalue specifies a new value to the default fill value attribute for an SDS array. It's only argument is `fill_val`, which specifies the new fill value. The fill value must be of the same number type as the array it's written to. To avoid conversion errors, use data-specific fill values instead of special architecture-specific values, such as infinity or Not-a-Number (or ***NaN***). Setting the fill value after data is written to the SDS will not update the fill values already written to the data set - it will only change the attribute.

The parameters for **DFSDsetfillvalue** are further defined in Table 11K below.

The **DFSDsetcal** routine creates a calibration record for a specified array and by doing so adds five attributes to the current data set. As the HDF library does not specifically apply calibration information to the data, **SDsetcal** can be called anytime before or after the data is written. **DFSDsetcal** has five arguments; `scale`, `scale_error`, `offset`, `off_err`, and `num_type`. The arguments `scale` and `offset` are defined as they are for the multifile SD API routines.

In addition to the `scale` and `offset`, **DFSDsetcal** also includes both a scale and offset error. The argument `scale_err` contains the potential error of the calibrated data due to scaling and `offset_err` contains the potential error for the calibrated data due to the offset. The `num_type` parameter specifies the number type of the uncalibrated data.

The parameters of **DFSDsetcal** are defined in the following table.

TABLE 11M **DFSDsetfillvalue, DFSDsetrange and DFSDsetcal Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	

DFSDsetfillvalue (dssfill)	label	char *	character* (*)	Label describing the data.
	unit	char *	character* (*)	Unit to be applied to the data.
	format	char *	character* (*)	Format to be applied in displaying the data.
	coordsys	char *	character* (*)	Coordinate system of the data set.
DFSDsetrange (dssrang)	max	VOIDP	<valid numeric data type>	Highest value in the selected range of data.
	min	VOIDP	<valid numeric data type>	Lowest value in the selected range of data.
DFSDsetcal (dsscal)	cal	float64	real*8	Calibration scale.
	cal_error	float64	real*8	Calibration scale error.
	off	float64	real*8	Uncalibrated offset.
	off_err	float64	real*8	Uncalibrated offset error.
	num_type	int32	integer	Number type of uncalibrated data.

EXAMPLE 3.

Assigning Predefined String Attributes to a File

The following examples demonstrate the steps necessary to assign predefined string attributes to the data set and stores the data set in the file "Example1.hdf". They create a string attribute using **DFSDsetdatastrs** and a value attribute using **DFSDsetrange**. It also demonstrates the use of **DFSDsetlengths** in altering the maximum string length from 255 characters to 50. It then writes the SDS array by calling **DFSDadddata**.

C:

```
#include "hdf.h"

/*
 *  Write an array of floating point values representing
 *  pressure in a 3x2x5 array.
 */

main( )
{

      float32 data[3][2][5];
      int32 dimsizes[3];
      float32 max, min;
      intn status, rank;
      int i, j, k;

      /* Set the rank and dimension sizes. */
      rank = 3;
      dimsizes[0] = 3;
      dimsizes[1] = 2;
      dimsizes[2] = 5;

      /* Set the dimensions, to define the beginning of a data set. */
      status = DFSDsetdims(rank, dimsizes);

      /* Set the maximum string length to 50. */
      status = DFSDsetlengths(50, 50, 50, 50);

      /* Define the attribute strings and values. */
      status = DFSDsetdatastrs("Pressure Data", "Millibars",
                "F5.5", "None");
      max = 1500.0;
      min = 0.0;
```

```
        status = DFSDsetrange(&max, &min);

        /* Set the rank to 3. */
        rank = 3;

        /* Calculate the data values. */
        for (i = 0; i < 3; i++)
          for (j = 0; j < 2; j++)
              for (k = 0; k < 5; k++)
                  data[i][j][k] = i*100.0 + j*10.0 + k;

        /* Write the data set and its attributes to file. */
        status = DFSDadddata("Example3.hdf", rank, dimsizes, data);

}
```

FORTRAN:

```
       PROGRAM SET ATTRIBS

       real*8 data(5, 2, 3), max, min, i, j, k
       integer*4 dimsizes(3)
       integer status, rank

       integer dsslens, dssdast, dssrang, dsadata
       integer dssdims

       character*13 label /"Pressure Data"/
       character*9 unit /"Millibars"/
       character*4 format /"F5.5"/
       character*4 coordsys /"None"/

C      Set the dimensions, to define the beginning of a data set.
       rank = 3
       dimsizes(1) = 5
       dimsizes(2) = 2
       dimsizes(3) = 3
       status = dssdims(rank, dimsizes)

C      Set the maximum string lengths to 50.
       status = dsslens(50, 50, 50, 50)

C      Define the attribute strings and values.
       status = dssdast(label, unit, format, coordsys)
       max = 1500.0
       min = 0.0
       status = dssrang(max, min)

C      Fill the data array with values.
       do 30 k = 1, 3
        do 20 j = 1, 2
         do 10 i = 1, 5
           data(i, j, k) = i*100.0 + j*10.0 + k
10       continue
20      continue
30     continue

C      Write the data set and its attributes to file.
       status = dsadata("Example3.hdf", rank, dimsizes, data)

       end
```

11.7.2 Reading DFSD Data Set Attributes

The DFSD interface provides two function calls for reading predefined data set attribute strings.

11.7.2.1 Reading Data Set Attributes: DFSDgetdatalen and DFSDgetdatastrs

DFSDgetdatalen returns the length of each string in the attribute. It is useful for determining the length of an attribute before reading it. **DFSDgetdatastrs** reads the label, unit, format, and coordinate system strings.

Attribute data is not read by **DFSDgetdatastrs** until the appropriate routine is called to read the array and its dimension record. If **DFSDgetdatastrs** and **DFSDgetrange** are not called, the array and its dimension record can be read without reading its associated data set attributes. It is also possible to read string and value attributes individually. As attribute data is not actually read by **DFSDgetdatastrs** or **DFSDgetrange**, these calls must be made before calling **DFSDgetdata**.

Reading the attributes of a data set involves the following steps:

1. Determine the length of each attribute string.
2. Read the attribute strings.
3. Read the maximum and minimum values.
4. Read the remainder of the data set.

To assign a predefined attribute to an HDF file, the following routines should be called:

C:
```
status = DFSDgetdatalen(label_len, unit_len, format_len,
                              coords_len);
status = DFSDgetdatastrs(label, unit, format, coordsys);
status = DFSDgetrange(max, min);
status = DFSDgetdata(filename, rank, dimsizes, data);
```

FORTRAN:
```
status = dsgdghaln(label_len, unit_len, format_len, coords_len)
status = dsgdast(label, unit, format, coordsys)
status = dsgrang(max, min)
status = dsgdata(filename, rank, dimsizes, data)
```

The parameters of **DFSDgetdatalen** and **DFSDgetdatastrs** are described in the following table.

TABLE 11N **DFSDgetdatalen and DFSDgetdatastrs Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFSDgetdatalen (dsgdaln)	label_len	intn *	integer	Length of any label string.
	unit_len	intn *	integer	Length of any unit string.
	format_len	intn *	integer	Length of any format string.
	coords_len	intn *	integer	Length of any coordinate system string.
DFSDgetdatastrs (dsgdast)	label	char *	character* (*)	Label describing the data.
	unit	char *	character* (*)	Unit applied to the data.
	format	char *	character* (*)	Format applied to the data.
	coordsys	char *	character* (*)	Coordinate system of the data set.

EXAMPLE 4. **Reading a Data Set and its Attribute Record**

These examples read the pressure data set and the dimension attribute record stored in the "Example1.hdf" file into the arrays pointed to by the `data`, `datalabel`, `dataunit`, `datafmt` and `coordsys` pointer variables. It assumes the dimension sizes and rank are correct and data strings are less than 10 characters long, with one additional character for the null termination.

C:

```
#include "hdf.h"

main( )
{

      intn rank, maxrank, status;
      int32 dimsizes[3];
      char datalabel[50], dataunit[50], datafmt[50], coordsys[50];
      float64 data[3][2][5];

      maxrank = 3;
      status = DFSDgetdims("Example3.hdf", &rank, dimsizes,
               maxrank);
      status = DFSDgetdatastrs(datalabel, dataunit, datafmt,
               coordsys);
      status = DFSDgetdata("Example3.hdf", rank, dimsizes, data);

}
```

FORTRAN:

```
       PROGRAM READ SD INFO

      integer dsgdata, dsgdast, dsgdims
      integer*4 dimsizes(3)
      integer status, rank, maxrank
      character*50 datalabel, dataunit, datafmt
      character*10 coordsys
      real*8 data(5, 2, 3)

      maxrank = 3
      status = dsgdims('Example3.hdf', rank, dimsizes, maxrank)
      status = dsgdast(datalabel, dataunit, datafmt, coordsys)
      status = dsgdata('Example3.hdf', rank, dimsizes, data)

      end
```

11.7.2.2 Reading the Value Attributes of a DFSD Data Set: DFSDgetfillvalue and DFSDgetcal

There are three routines in the DFSD interface that retrieve the fill value, range and calibration information of a data set array: **DFSDgetfillvalue**, **DFSDgetrange** and **DFSDgetcal**.

The syntax of these routines are as follows:

C:

```
status = DFSDgetfillvalue(sds_id, fill_val);
status = DFSDgetrange(max, min);
status = DFSDgetcal(cal, cal_err, offset, offset_err, num_type);
```

FORTRAN:

```
status = dsgfill(fill_value)
status = dsgrang(max, min)
status = dsadata(cal, cal_err, offset, offset_err, num_type)
```

DFSDgetfillvalue has two arguments; `sds_id` and `fill_val`. The `sds_id` is the data set identifier and `fill_val` is the space allocated to store the fill value.

The maximum range of values in the data set isn't automatically stored with the data set data; it is explicitly stored through a call to **DFSDgetrange**. The defined range of values can be less than the actual range of values stored in the data set. The value of the `max` parameter is the maximum value of the defined range and the value of the `min` parameter is the minimum value. These values must be of the same number type as the values stored in the data array. In C, the `max` and `min` parameters are indirect pointers specifying the range values, while in Fortran-77 they are variables set to the range values.

DFSDgetcal reads the calibration record of the current data set, if one exists. Each of the parameters of **DFSDgetcal** correspond to the five elements of the calibration record; - four 64-bit floating-point integers followed by a 32-bit integer. The `cal`, `offset`, `offset_err` and `cal_err` parameters are defined as they are in the multifile SD API. This calibration record exists for information only.

The parameters for **DFSDgetfillvalue**, **DFSDgetcal** and **DFSDgetrange** are defined in the following table.

TABLE 11O **DFSDgetfillvalue, DFSDgetcal and DFSDgetrange Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFSDgetfillvalue (dsgfill)	sds_id	int32	integer	Data set identifier.
	fill_val	VOIDP	<valid numeric data type>	Buffer for the fill value.
DFSDgetcal (dsgcal)	cal	float64 *	real*8	Calibration factor.
	cal_err	float64 *	real*8	Calibration error.
	offset	float64 *	real*8	Uncalibrated offset.
	offset_err	float64 *	real*8	Uncalibrated offset error.
	num_type	`int32 *`	`integer`	Type of the uncalibrated data.
DFSDgetrange (dsgrang)	max	VOIDP	<valid numeric data type>	Highest value of the selected range.
	min	VOIDP	<valid numeric data type>	Lowest value of the selected range.

11.7.3 Writing the Dimension Attributes of a DFSD SDS

Dimension attributes are described in Chapter 3, titled *Scientific Data Sets (SD API)*.

11.7.3.1 Writing the String Attributes of a Dimension: DFSDsetlengths and DFSDsetdimstrs

The DFSD interface provides two routines for creating dimension string attributes: **DFSDsetlengths** and **DFSDsetdimstrs**. **DFSDsetlengths** overwrites the default string length and **DFSDsetdimstrs** is defines the string text. **DFSDsetdatalengths** and **DFSDsetdimstrs** are optional and must precede calls to **DFSDadddata** or **DFSDputdata**.

Predefined dimension string attributes are limited to one per dimension and contain the following:

- ***Format attributes*** specify the format to use when displaying values for the dimension.
- ***Label attributes*** contain dimension names.
- ***Unit attributes*** identify the unit of measurement associated with the dimension.

To assign a predefined attribute to a dimension, the following routines should be called:

C:
```
status = DFSDsetlengths(label_len, unit_len, format_len,
                        coords_len);
status = DFSDsetdimstrs(label, unit, format);
status = DFSDadddata(filename, rank, dimsizes, data);
```

FORTRAN:
```
status = dsslens(label_len, unit_len, format_len, coords_len)
status = dssdist(label, unit, format)
status = dsadata(filename, rank, dimsizes, data)
```

DFSDsetlengths has four arguments: `label_len`, `unit_len`, `format_len`, and `coords_len`. Each parameter specifies the maximum length of the string that defines the label, unit, format, and coordinate system. As mentioned earlier in this chapter, attribute lengths seldom need to be reset..

DFSDsetdimstrs also has four arguments; `dim`, `label`, `unit`, and `format`. The parameter `dim = 1` for the first dimension, `dim = 2` for the second dimension, etc.. To avoid assigning a string to the coordinate length, pass `NULL` in the appropriate parameter. **DFSDsetdimstrs** writes null-terminated strings to a file.

The parameters for **DFSDsetlengths** and **DFSDsetdimstrs** are further defined in the following table.

TABLE 11P **DFSDsetlengths and DFSDsetdimstrs Parameter List**

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFSDsetlengths (dsslen)	label_len	intn	integer	Maximum length of any label string.
	unit_len	intn	integer	Maximum length of any unit string.
	format_len	intn	integer	Maximum length of any format string.
	coords_len	intn	integer	Maximum length of any coordinate system string.
DFSDsetdimstrs (dssdist)	dim	intn	integer	Dimension of the attribute strings.specified by the remaining three parameters
	label	char *	character* (*)	Label describing the data.
	unit	char *	character* (*)	Unit to be applied to the data.
	format	char *	character* (*)	Format to be applied in displaying the data.

11.7.3.2 Writing a Dimension Scale of a DFSD SDS: DFSDsetdimscale

The syntax of the two routines needed to write a dimension scale is the following:

C:
```
status = DFSDsetdimscale(dim, dimsize, scale);
status = DFSDadddata(filename, rank, dimsizes, data);
```

FORTRAN:
```
status = dssdisc(dim, dimsize, scale)
status = dsadata(filename, rank, dimsizes, data)
```

DFSDsetdimscale has three arguments; `dim`, `dimsize`, and `scale`. These arguments identify the dimension, specify its size, and assign a value to each of its grid points. The parameter `dim = 1` for the first dimension, and `dim = 2` for the second dimension. The `dimsize` argument must contain a value equal to the dimension it describes in order for the scale to be applied correctly.

The parameters of **DFSDsetdiscale** are further described in the following table.

TABLE 11Q

DFSDsetdimscale Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFSDsetdimscale (dssdisc)	dim	intn	integer	Dimension of the current scale.
	dim_size	int32	integer	Size of the current scale.
	scale	VOIDP	<valid numeric data type>	Values of the current scale.

11.7.4 Reading the Dimension Attributes of a DFSD SDS

The DFSD interface provides three routines for reading dimension attributes: **DFSDgetdimlen**, **DFSDgetdimstrs** and **DFSDgetdimscale**. **DFSDgetdimlen** returns the string length for each string in the attribute record. It is a useful routine to call before reading an attribute. **DFSDget-dimstrs** and **DFSDgetdimscale** are used as instructions for reading the dimension attributes. **DFSDgetdimstrs** reads the dimension strings and **DFSDgetdimscale** reads the dimension scale. By avoiding calls to **DFSDgetdimstrs** and **DFSDgetdimscale**, it is possible to read an array and its dimension record without reading the data set attributes associated with it. It is also possible to omit one function call in order to read one attribute without the other. **DFSDgetdimstrs** and **DFS-Dgetdimscale** must be called before **DFSDgetdata**.

Reading data set attributes involves the following steps:

1. Determine the length of each attribute string.
2. Read the attribute strings.
3. Read the scale values.
4. Read the remainder of the data set.

To assign a predefined attribute to an HDF file, the following routines must be called:

C:
```
status = DFSDgetdimlen(label_len, unit_len, format_len,
                             coords_len);
status = DFSDgetdimstrs(label, unit, format);
status = DFSDgetdimscale(dim, dim_size, scale);
status = DFSDgetdata(filename, rank, dimsizes, data);
```

FORTRAN:
```
status = dsgdiln(label_len, unit_len, format_len, coords_len)
status = dsgdist(label, unit, format)
status = dsgdisc(dim, dim_size, scale)
status = dsgdata(filename, rank, dimsizes, data)
```

The parameters for **DFSDgetdimlen**, **DFSDgetdimstrs** and **DFSDgetdimscale** are described in the following table.

TABLE 11R

DFSDgetdimlen, DFSDgetdimstrs and DFSDgetdimscale Parameter List

Routine Name (Fortran-77)	Parameter	Data Type		Description
		C	Fortran-77	
DFSDgetdimlen (dsgdiln)	dim	intn	integer	Dimension of the string attributes describe.
	label_len	intn *	integer	Length of the label attribute string.
	unit_len	intn *	integer	Length of the unit attribute string.
	format_len	intn *	integer	Length of the format attribute string.

DFSDgetdimstrs (dsgdist)	dim	intn	integer	Dimension the string attributes describe.
	label	char *	character* (*)	Label of the dimension.
	unit	char *	character* (*)	Unit to be applied to this dimension.
	format	char *	character* (*)	Format to be applied when displaying the scale.
DFSDgetdimscale (dsgdisc)	dim	intn	integer	Dimension the current scale is attached to
	dim_size	int32	integer	Size of the current scale.
	scale	VOIDP	<valid numeric data type>	Values of the current scale.

Chapter 12

Error Reporting

12.1 Chapter Overview

This chapter describes the main error reporting routines designed for general HDF use and the types of errors handled by the error reporting API and the general structure of the API.

12.2 The HDF Error Reporting API

The HDF error reporting API consists of routines that query error stack information, the names of which are prefaced by "HE". They are described briefly in Table 12A. Some are primarily for use by HDF developers while others are available to HDF users. In this chapter, three error reporting functions are covered: **HEprint**, **HEvalue** and **HEstring**. Note that only one C error reporting routine has a Fortran-77 counterpart: **HEprint**.

TABLE 12A **Error Reporting Routine List**

Category	Routine Name		Description
	C	Fortran-77	
Error Reporting	HEprint	heprnt	Prints the errors on the error stack to a specified file.
	HEstring	None	Returns the error message associated with an error code.
	HEvalue	None	Returns the nth most recent error reported.

12.3 Error Reporting in HDF

Most HDF error reporting routines return `FAIL` (or `-1`) if the operation is successful and `SUCCEED` (or `0`) otherwise. Each time a `FAIL` code is returned one or more error codes are pushed onto the error code stack. The following pseudo-code will demonstrate the two methods commonly used to access and print the contents of this stack.

A list of error codes is included at the end of this chapter.

```
if (<general HDF function() >= FAIL) {
    <HDF error reporting API routines>
}
```

OR

```
status = <general HDF function( );
if (status == FAIL) {
    <HDF error reporting API routines>
}
```

12.3.1 Writing Errors to a File: HEprint

HEprint writes the errors on the stack to the specified file. There are four sections of an **HEprint** error report:

1. A description of the error.
2. The routine in which the error was detected.
3. The source file in which the error was detected.
4. The line number in which the error was detected.

The syntax for **HEprint** is as follows:

C: `HEprint(stream, level);`

FORTRAN: `status = heprnt(level)`

The `stream` parameter is a UNIX file handle indicating the output stream the error information will be written to. The `level` parameter specifies the amount of error information to report. In Fortran-77, **HEprint** always writes to the standard error stream, or `stderr`, therefore the only parameter is `level`.

Errors are written in sequential order starting from the bottom of the stack. Consequently, specifying a `level` parameter value of 1 will write the first error that occurred, or the first error pushed onto the stack. Specifying a `level` parameter of value 0 will write all errors on the stack to the specified file. For example, the following C code will write all errors on the stack to the file named “errors”.

```
f = fopen(“errors”, “w”);
HEprint(f, 0);
```

As an example of the output of **HEprint**, suppose an attempt is made to open an nonexistent file with **Hopen**. Calling `HEprint(stdout, 0)` or `heprnt(0)` will produce the following output:

```
HDF error: <error opening file>
    Detected in Hopen() [hfile.c line 305]
```

12.3.2 Returning the Code of the Nth Most Recent Error: HEvalue

HEvalue returns the error code for the nth most recent error and is only available as a C routine. The `error_stack_offset` parameter specifies the number of errors to regress from the top of the error stack.

The syntax for **HEvalue** is as follows:

C:
```
status = HEvalue(error_stack_offset);
```

12.3.3 Returning the Description of an Error Code: HEstring

HEstring returns the error description associated with the error code specified by the `error_code` parameter as a character string. As with **HEvalue**, **HEstring** is only available in the C HDF library.

The syntax for **HEstring** is as follows:

C:
```
error_string = HEstring(error_code);
```

EXAMPLE 1. **Writing Errors to a Console Window**

The following C code fragment will copy errors from the stack to a console window.

C:
```
#include "hdf.h"

main( )
{

int32 i, e;
const char *str;

...

i = 0;
while ((e = HEvalue(i)) != DFE_NONE) {
        str = HEstring(e);
        <device-specific code to print the string to a console>
        i++

...

}
```

TABLE 12B **HDF Error Codes**

Error Code	Code Definition
DFE_NONE	No error.
DFE_FNF	File not found.
DFE_DENIED	Access to file denied.
DFE_ALROPEN	File already open.
DFE_TOOMANY	Too many AID's or files open.
DFE_BADNAME	Bad file name on open.
DFE_BADACC	Bad file access mode.
DFE_BADOPEN	Miscellaneous open error.
DFE_NOTOPEN	File can't be closed because it hasn't been opened.
DFE_CANTCLOSE	`fclose` error

Error Code	Code Definition
DFE_READERROR	Read error.
DFE_WRITEERROR	Write error.
DFE_SEEKERROR	Seek error.
DFE_RDONLY	File is read only.
DFE_BADSEEK	Attempt to seek past end of element.
DFE_PUTELEM	`Hputelement` error.
DFE_GETELEM	`Hgetelement` error.
DFE_CANTLINK	Cannot initialize link information.
DFE_CANTSYNC	Cannot syncronize memory with file.
DFE_BADGROUP	Error from `DFdiread` in opening a group.
DFE_GROUPSETUP	Error from `DFdisetup` in opening a group.
DFE_PUTGROUP	Error on putting a tag/reference number pair into a group.
DFE_GROUPWRITE	Error when writing group contents.
DFE_DFNULL	Data file reference is a null pointer.
DFE_ILLTYPE	Data file contains an illegal type: internal error.
DFE_BADDDLIST	The DD list is non-existent: internal error.
DFE_NOTDFFILE	The current file is not an HDF file and it is not zero length.
DFE_SEEDTWICE	The DD list already seeded: internal error.
DFE_NOSUCHTAG	No such tag in the file: search failed.
DFE_NOFREEDD	There are no free DD's left: internal error.
DFE_BADTAG	Illegal WILDCARD tag.
DFE_BADREF	Illegal WILDCARD reference number.
DFE_NOMATCH	No DDs (or no more DDs) that match the specified tag/reference number pair.
DFE_NOTINSET	Warning: Set contained unknown tag. Ignored.
DFE_BADOFFSET	Illegal offset specified.
DFE_CORRUPT	File is corrupted.
DFE_NOREF	No more reference numbers are available.
DFE_DUPDD	The new tag/reference number pair has been allocated.
DFE_CANTMOD	Old element doesn't exisr. Cannot modify.
DFE_DIFFFILES	Attempt to merge objects in different files.
DFE_BADAID	An invalid AID was received.
DFE_OPENAID	Active AIDs still exist.
DFE_CANTFLUSH	Cannot flush DD back to file.
DFE_CANTUPDATE	Cannot update the DD block.
DFE_CANTHASH	Cannot add a DD to the hash table.
DFE_CANTDELDD	Cannot delete a DD in the file.
DFE_CANTDELHASH	Cannot delete a DD from the hash table.
DFE_CANTACCESS	Cannot access specified tag/reference number pair.
DFE_CANTENDACCESS	Cannot end access to data element.
DFE_TABLEFULL	Access table is full.
DFE_NOTINTABLE	Cannot find element in table.
DFE_UNSUPPORTED	Feature not currently supported.
DFE_NOSPACE	`malloc` failed.
DFE_BADCALL	Routine calls were in the wrong order.
DFE_BADPTR	NULL pointer argument was specified.
DFE_BADLEN	Invalid length was specified.
DFE_NOTENOUGH	Not enought space for the data.

Error Code	Code Definition
DFE_NOVALS	Values were not available.
DFE_ARGS	Invalid arguments passed to the routine.
DFE_INTERNAL	Serious internal error.
DFE_NORESET	Too late to modify this value.
DFE_GENAPP	Generic application level error.
DFE_UNINIT	Interface was not initialized correctly.
DFE_CANTINIT	Cannot initialize the interface the operation requires.
DFE_CANTSHUTDOWN	Cannot shut down the interface the operation requires.
DFE_BADDIM	Negative number of dimensions, or zero dimensions, was specified.
DFE_BADFP	File contained an illegal floating point number.
DFE_BADDATATYPE	Unknown or unavailable data type was specified.
DFE_BADMCTYPE	Unknown or unavailable machine type was specified.
DFE_BADNUMTYPE	Unknown or unavailable number type was specified.
DFE_BADORDER	Unknown or illegal array order was specified.
DFE_RANGE	Improper range for attempted access.
DFE_BADCONV	Invalid data type conversion was specified..
DFE_BADTYPE	Incompatible types were specified.
DFE_BADSCHEME	Unknown compression scheme was specified.
DFE_BADMODEL	Invalid compression model was specified.
DFE_BADCODER	Invalid compression encoder was specified.
DFE_MODEL	Error in the modeling layer of the compression operation.
DFE_CODER	Error in the encoding layer of the compression operation.
DFE_CINIT	Error in encoding initialization.
DFE_CDECODE	Error in decoding compressed data.
DFE_CENCODE	Error in encoding compressed data.
DFE_CTERM	Error in encoding termination.
DFE_CSEEK	Error seeking in an encoded dataset.
DFE_MINIT	Error in modeling initialization.
DFE_COMPINFO	Invalid compression header.
DFE_CANTCOMP	Cannot compress an object.
DFE_CANTDECOMP	Cannot decompress an object.
DFE_NODIM	A dimension record was not associated with the image.
DFE_BADRIG	Error processing a RIG.
DFE_RINOTFOUND	Cannot find raster image.
DFE_BADATTR	Invalide attribute.
DFE_BADTABLE	The nsdg table has incorrect information.
DFE_BADSDG	Error in processing an SDG.
DFE_BADNDG	Error in processing an NDG.
DFE_VGSIZE	Too many elements in the vgroup.
DFE_VTAB	Element not in `vtab[].`
DFE_CANTADDELEM	Cannot add the tag/reference number pair to the vgroup.
DFE_BADVGNAME	Cannot set the vgroup name.
DFE_BADVGCLASS	Cannot set the vgroup class.
DFE_BADFIELDS	Invalid fields string passed to vset routine.
DFE_NOVS	Cannot find the vset in the file.
DFE_SYMSIZE	Too many symbols in the users table.
DFE_BADATTACH	Cannot write to a previously attached vdata.

Error Code	Code Definition
DFE_BADVSNAME	Cannot set the vdata name.
DFE_BADVSCLASS	Cannot set the vdata class.
DFE_VSWRITE	Error writing to the vdata.
DFE_VSREAD	Error reading from the vdata.
DFE_BADVH	Error in the vdata header.
DFE_VSCANTCREATE	Cannot create the vdata.
DFE_VGCANTCREATE	Cannot create the vgroup.
DFE_CANTATTACH	Cannot attach to a vdata or vset.
DFE_CANTDETACH	Cannot detach a vdata or vset with write access.
DFE_BITREAD	A bit read error occurred.
DFE_BITWRITE	A bit write error occurred.
DFE_BITSEEK	A bit seek error occurred.
DFE_TBBTINS	Failed to insert the element into tree.
DFE_BVNEW	Failed to create a bit vector.
DFE_BVSET	Failed when setting a bit in a bit vector.
DFE_BVGET	Failed when getting a bit in a bit vector.
DFE_BVFIND	Failed when finding a bit in a bit vector.

Chapter 13

HDF Performance Issues

13.1 Chapter Overview and Introduction

This chapter describes many of the concepts the HDF user should understand to gain better performance from their applications that use the HDF library. It also covers many of the ways in which HDF can be used to cause impaired performance and methods for correcting these problems.

As stated earlier in this manual, HDF has been designed to be very general-purpose, and it has been used in many different applications involving scientific data. Each application has its own set of software and hardware resource constraints that will affect performance in a different way, and to a different extent, from the resource constraints in other applications.

Therefore, it is impossible to outline *all* of the performance issues that may relate to a particular application of HDF. However, this chapter should give the reader sufficient knowledge of the most common performance issues encountered by the HDF Group. This knowledge should enable the reader to explore different ways of storing data on their native platforms for the purpose of increasing library performance.

Future versions of this chapter will include additional possibilities of performance enhancement as they are discovered.

13.2 Examples of HDF Performance Enhancement

In this section, four pairs of HDF object models along with their C implementations will be presented. Each pair will illustrate a specific aspect of HDF library performance as it relates to scientific data sets. They will be employed here as general pointers on how to model scientific data sets for optimized performance.

In developing and testing these examples, the Sun Solaris OS version supported by HDF version 4.1 release 1 was used. Version 2.0 of the Quantify performance profiler was used to measure the relative differences in library performance between the SDS models in each pair. It should be noted that, while the examples reliably reflect which SDS configurations result in better performance, the specifics of how much performance will be improved depend on many factors such as OS configuration, compiler used and profiler used. Therefore, any specific measurements of performance mentioned in the chapter should be interpreted only as general indicators.

The C programs that were used as the basis of this section can be obtained from the HDF ftp server at hdf.ncsa.uiuc.edu in the /pub/dist/HDF/HDF4.1r1/Performance directory. These are provided in the event the reader wishes to verify or modify these examples on their own system.

The reader should keep in mind that the following examples have been designed for illustrative purposes only, and should not be considered as real-world examples. It is expected that the reader will apply the library performance concepts covered by these examples to their specific usage of the HDF library.

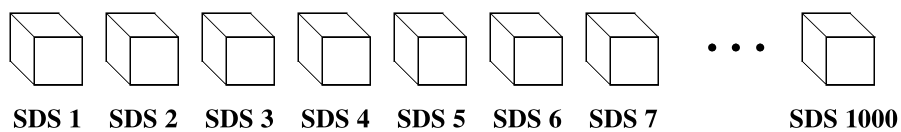
13.2.1 One Large SDS Versus Several Smaller SDSs

The scientific data set is an example of what in HDF parlance is referred to as a *primary object*. The primary objects accessed and manipulated by the HDF library include, beside scientific data sets, raster images, annotations, vdatas and vgroups. Each primary object has *metadata*, or data describing the data, associated with it. Refer to the *HDF Specifications Manual* for a description of the components of this metadata and how to calculate its size on disk.

An opportunity for performance enhancement can exist when the size of the metadata far exceeds the size of the data described by the metadata. In this situation, more CPU time and disk space will be used to maintain the metadata than the data contained in the SDS. Consolidating the data into fewer, or even one, SDS can increase performance.

To illustrate this, consider 1,000 1 *x* 1 *x* 1 element scientific data sets of 32-bit floating-point numbers. No user-defined dimension, dimension scales or fill values have been defined or created.

FIGURE 13a **1,000 1 *x* 1 *x* 1 Element Scientific Data Sets**



In this example, 1,000 32-bit floating-point numbers are first buffered in-core, then written to the SDS.

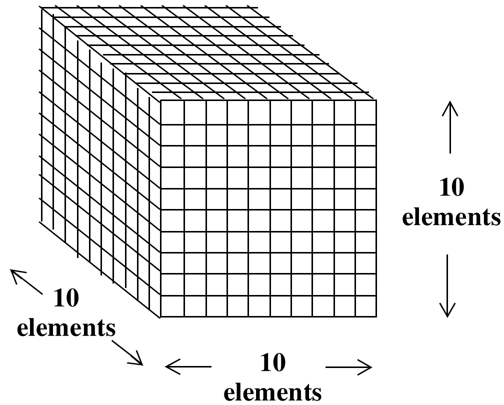
In Table 13A, the results of this operation are reflected in two metrics: the total number of CPU cycles used by the example program, and the size of the HDF file after the write operation.

TABLE 13A **Results of the Write Operation to 1,000 1 *x* 1 *x* 1 Element Scientific Data Sets**

Total Number of CPU Cycles	Size of the HDF File (in bytes)
136,680,037	896,803

Now the 1,000 32-bit floating point numbers that were split into 1,000 SDSs are combined into one 10 *x* 10 *x* 10 element SDS. This is illustrated in the following figure.

FIGURE 13b **One 10 *x* 10 *x* 10 Element Scientific Data Set**



As with the last example, 1,000 32-bit floating-point numbers are first buffered in-core, then written to the SDS. The following table contains the performance metrics of this operation.

TABLE 13B **Results of the Write Operation to One 10 *x* 10 *x* 10 Element Scientific Data Set**

Total Number of CPU Cycles	Size of the HDF File (in bytes)
205,201	7,258

It is apparent from these results that merging the data into one scientific data set results in a substantial increase in I/O efficiency - in this case, a 99.9% reduction in total CPU load. In addition, the size of the HDF file is dramatically reduced by a factor of more than 100, even through the amount of SDS data stored is the same.

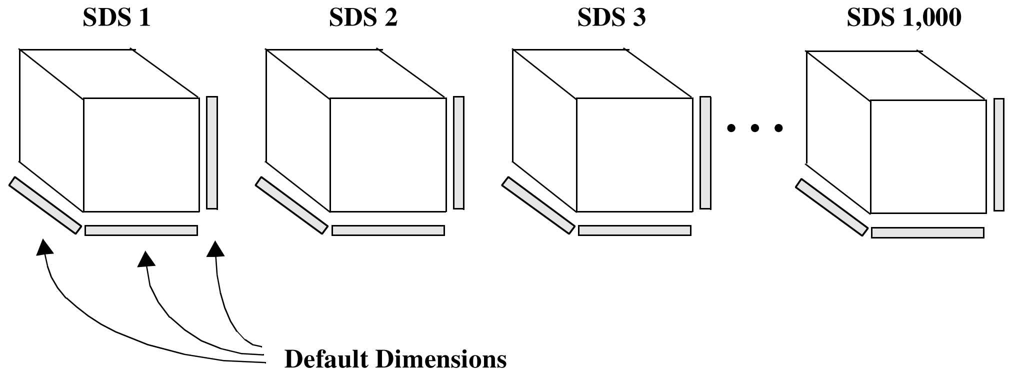
The extent to which the data consolidation described in this section should be done is dependent on the specific I/O requirements of the HDF user application.

13.2.2 Sharing Dimensions Between Scientific Data Sets

When several scientific data sets have dimensions of the same length, name and data type, they can share these dimensions to reduce storage overhead and CPU cycles in writing out data.

To illustrate this, again consider the example of 1,000 1 *x* 1 *x* 1 scientific data sets of 32-bit floating point numbers. Three dimensions are attached by default to each scientific data set by the HDF library. The HDF library assigns each of these dimensions a default name prefaced by "fakeDim". See Chapter 3 of the *HDF User's Guide*, titled *Scientific Data Sets (SD SDS)*, for a specific explanation of default dimension naming conventions.

FIGURE 13c **1,000 1 *x* 1 *x* 1 Element Scientific Data Sets**



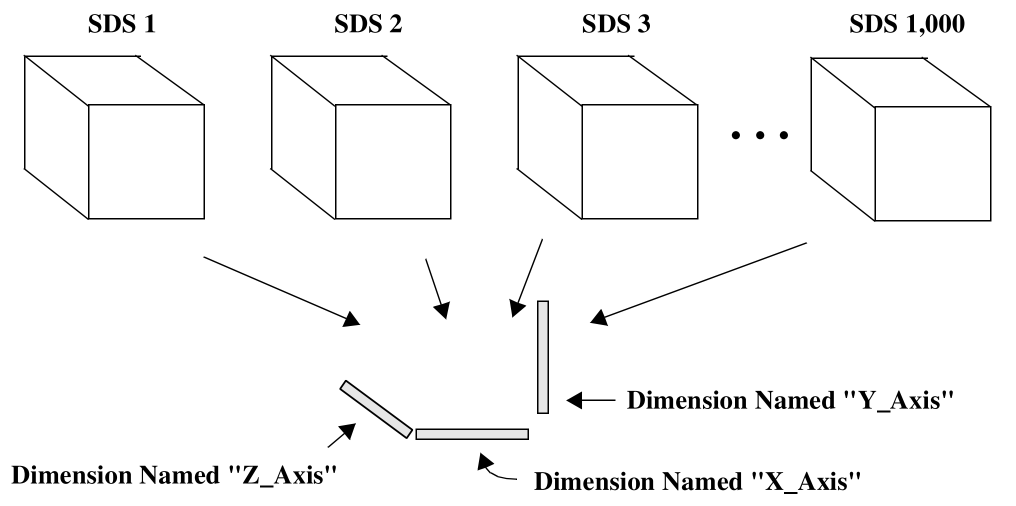
One 32-bit floating point number is written to each scientific data set. The following table lists the performance metrics of this operation.

TABLE 13C **Results of the Write Operation to 1,000 1 *x* 1 *x* 1 Element Scientific Data Sets**

Total Number of CPU Cycles	Size of the HDF File (in bytes)
136,680,037	896,803

Now consider the 1,000 SDSs described previously in this section. In this case, the 1,000 SDSs share the program-defined "X_Axis", "Y_Axis" and "Z_Axis" dimensions as illustrated in the following figure.

FIGURE 13d **1,000 1 *x* 1 *x* 1 Element Scientific Data Sets Sharing Dimensions**



The performance metrics that result from writing one 32-bit floating-point number to each dataset are in the following table.

TABLE 13D

Results of the Write Operation to 1,000 1 *x* 1 *x* 1 SDSs with Shared Dimensions

Total Number of CPU Cycles	Size of the HDF File (in bytes)
24,724,384	177,172

A 82% performance improvement in this example program can be seen from the information in this table, due to the fewer write operations involved in writing dimension data to shared dimensions. Also, the HDF file is significantly smaller in this case, due to the smaller amount of dimension data that is written.

13.2.3 Setting the Fill Mode

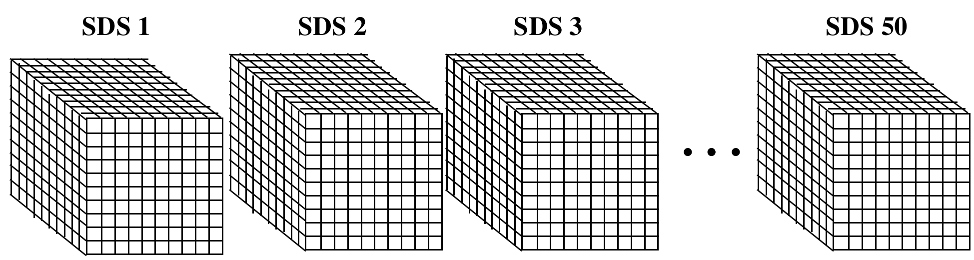
When a scientific data set is created, the default action of the HDF library is to fill every element with the default fill value. This action can be disabled, and reenabled once it has been disabled, by a call to the **SDsetfillmode** routine.

The library's default writing of fill values can degrade performance when, after the fill values have been written, every element in the dataset is written to again. This operation involves writing every element in the SDS twice. This section will demonstrate that disabling the initial fill value write operation by calling **SDsetfillmode** can improve library performance.

Consider 50 10 *x* 10 *x* 10 scientific data sets of 32-bit floating-point numbers.

FIGURE 13e

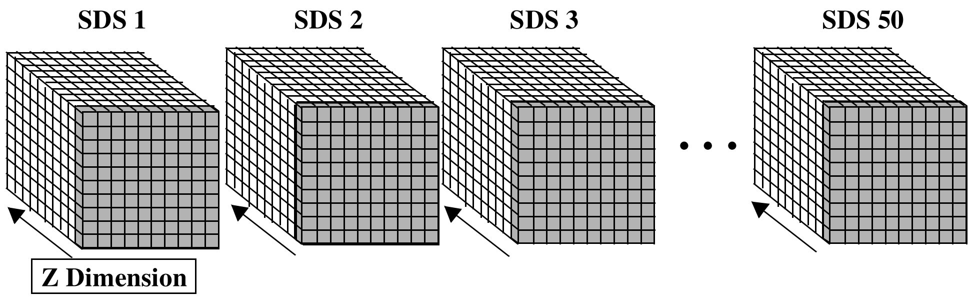
50 10 *x* 10 *x* 10 Element Scientific Data Sets



By default, the fill value is written to every element in all 50 SDSs. The contents of a two-dimensional buffer containing 32-bit floating-point numbers is then written to these datasets. The way these two-dimensional slices are written to the three-dimensional SDSs is illustrated in the following figure. Each slice (represented by each shaded area in the figure) is written along the third dimension of each SDS, or if the dimensions are related to a Cartesian grid, the z-dimension, until the entire SDS is filled.

FIGURE 13f

Writing to the 50 10 *x* 10 *x* 10 Element Scientific Data Sets



It should be noted that the reason each SDS isn't rewritten to in one write operation is because the HDF library will detect this and automatically disable the initial write of the fill values as a performance-saving measure. Hence, the partial writes in two-dimensional slabs.

The following table shows the number of CPU cycles needed in our tests to perform this write operation with the fill value write enabled. The "Size of the HDF File" metric has been left out of this table, because it won't change substantially regardless of whether the default fill value write operation is enabled.

TABLE 13E

Results of the Write Operation to the 50 10 *x* 10 *x* 10 SDSs with the Fill Value Write Enabled

Total Number of CPU Cycles
584,956,078

The following table shows the number of CPU cycles needed to perform the same write operation with the fill value write disabled.

TABLE 13F

Results of the Write Operation to the 50 SDSs with the Fill Value Write Disabled

Total Number of CPU Cycles
390,015,933

The information in these tables demonstrate that eliminating the I/O overhead of the default fill value write operation when an entire SDS is rewritten to results in a substantial reduction of the CPU cycles needed to perform the operation - in this case, a reduction of 33%.

13.2.4 Disabling "Fake" Dimension Scale Values in Large One-Dimensional Scientific Data Sets

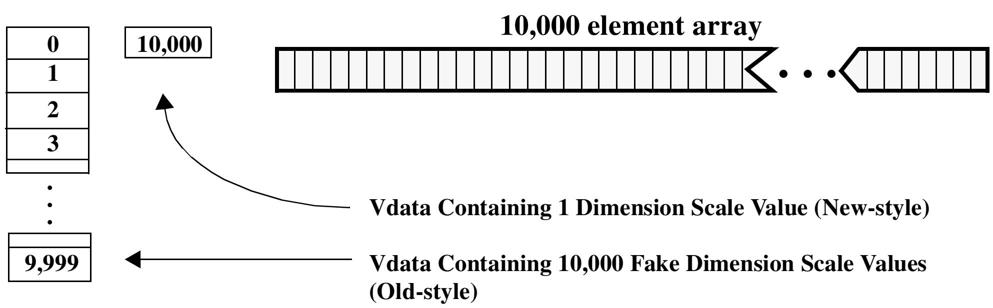
In versions 4.0 and earlier of the HDF library, dimension scales were represented by a vgroup containing a vdata. This vdata consisted of as many records as there are elements along the dimension. Each record contained one number which represented each value along the dimension scale, and these values are referred to as "fake" dimension scale values.

In HDF version 4.0 a new representation of the dimension scale was implemented alongside the old one - a vdata containing only one value representing the total number of values in the dimension scale. In version 4.1 release 1, this representation was made the default. A "compatible" mode is also supported where both the older and newer representations of the dimension scale are written to file.

In the earlier representation, a substantial amount of I/O overhead is involved in writing the fake dimension scale values into the vdata. When one of the dimensions of the SDS array is very large, performance can be improved, and the size of the HDF file can be reduced, if the old representation of dimension scales is disabled by a call to the **SDsetdimval_comp** routine. The examples in this section will illustrate this.

First, consider one 10,000 element array of 32-bit floating point numbers, as shown in the following figure. Both the new and old dimension scale representations are enabled by the library.

FIGURE 13g **One 10,000 Element Scientific Data Set with Old- and New-Style Dimension Scales**



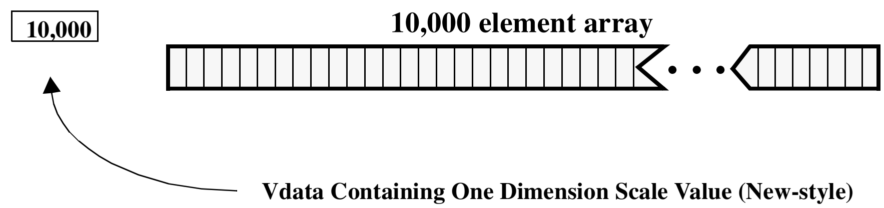
10,000 32-bit floating-point numbers are buffered in-core, then written to the scientific data set. In addition, 10,000 integers are written to the SDS as dimension scale values. The following table contains the results of this operation from our tests.

TABLE 13G **Results of the SDS Write Operation with the New and Old Dimension Scales**

Total Number of CPU Cycles	Size of the HDF File (in bytes)
439,428	82,784

Now consider the same SDS with the fake dimension scale values disabled. The following figure illustrates this.

FIGURE 13h **One 10,000 Element Scientific Data Set With the Old-Style Dimension Scale Disabled**



The following table contains the performance metrics of this write operation.

TABLE 13H **Results of the SDS Write Operation With Only the New Dimension Scale**

Total Number of CPU Cycles	Size of the HDF File
318,696	42,720

The old-style dimension scale is not written to the HDF file, which results in the size of the file being reduced by nearly 50%. There is also a marginal reduction in the total number of CPU cycles.

13.3 Data Chunking

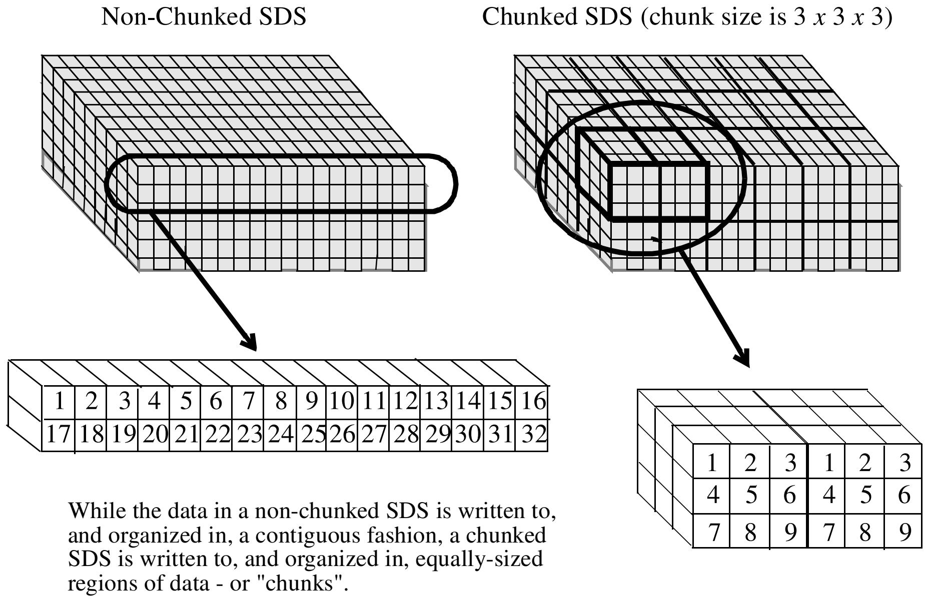
13.3.1 What is Data Chunking?

Data chunking is a method of organizing data within an SDS where data is stored in "chunks" of a predefined size, rather than contiguously by array element. Its two-dimensional instance is sometimes referred to as "data tiling". Data chunking is generally beneficial to I/O performance in very large arrays - i.e., arrays with thousands of rows and columns.

If correctly applied, data chunking may reduce the number of seeks through the SDS data array to find the data to be read or written, thereby improving I/O performance. However, it should be remembered that data chunking, if incorrectly applied, can significantly *reduce* the performance of reading and/or writing to an SDS. Knowledge of how chunked SDSs are created and accessed and application-specific knowledge of how data is to be read from the chunked SDSs are necessary in avoiding situations where data chunking works against the goal of I/O performance optimization.

The following figure illustrates the difference between a non-chunked SDS and a chunked SDS.

FIGURE 13i **Comparison Between Chunked and Non-Chunked Scientific Data Sets**



13.3.2 Writing Concerns and Reading Concerns in Chunking

There are issues in working with chunks that are related to the reading process and others that are related to the writing process.

Specifically, the issues that affect the process of reading from chunked SDSs are

- Compression
- Subsetting
- Chunk sizing
- Chunk cache sizing

The issues that affect the process of writing to chunked SDSs are

- Compression
- Chunk cache sizing

13.3.3 Chunking without Compression

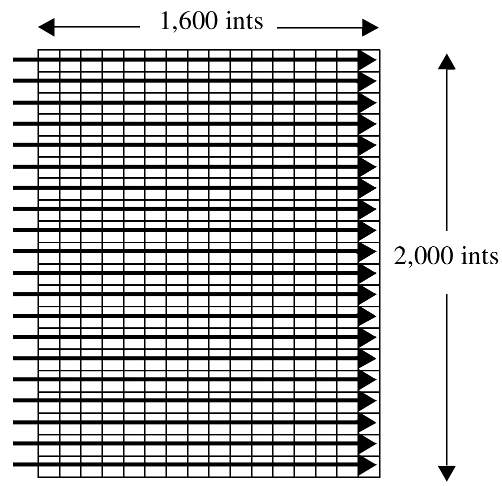
Accessing Subsets According to Storage Order

The main consideration to keep in mind when subsetting from chunked and non-chunked SDSs is that if the subset can be accessed in the same order as it was stored, subsetting will be efficient. If not, subsetting may result in less-than-optimal performance considering the number of elements to be accessed.

To illustrate this, the instance of subsetting in non-chunked SDSs will first be described. Consider the example of a non-chunked, two-dimensional, 2,000 *x* 1,600 SDS array of integer data. The following figure shows how this array is filled with data in a row-wise fashion. (Each square in the array shown represents 100 *x* 100 integers.)

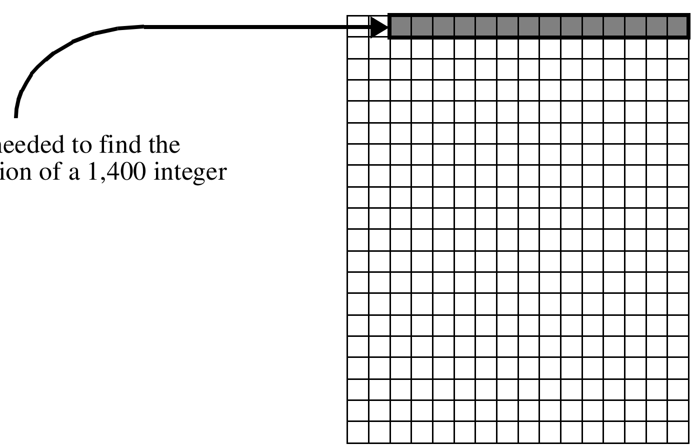
FIGURE 13j **Filling a Two-Dimensional Array With Data Using Row-Major Ordering**

In C, a two dimensional array is filled row-wise.



The most efficient way an application can read a row of data, or a portion of a row, from this array, is a contiguous, row-wise read of array elements. This is because this is the way the data was originally written to the array. Only one seek is needed to perform this. (See Figure 13k.)

FIGURE 13k **Number of Seeks Needed to Access a Row of Data in a Non-Chunked SDS**



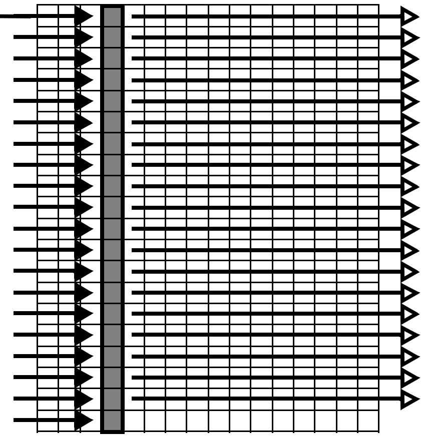
One seek is needed to find the starting location of a 1,400 integer row of data.

If the subset of data to be read from this array is one 2,000 integer *column*, then 2,000 seeks will be required to complete the operation. This is the most inefficient method of reading this subset as nearly all of the array locations will be accessed in the process of seeking to a relatively small number of target locations.

FIGURE 13l

Number of Seeks Needed to Access a Column of Data in a Non-Chunked SDS

2,000 seeks are needed to find the starting location of each element in a 2,000 integer column of data. (Each arrow represents 100 seeks.)

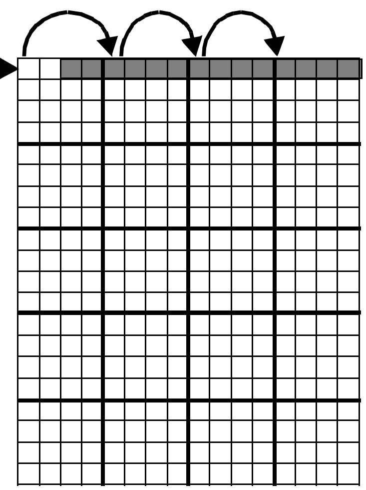


Now suppose this SDS is chunked, and the chunk size is 400 *x* 400 integers. A read of the aforementioned row is performed. In this case, four seeks are needed to read all of the chunks that contain the target locations. This is less efficient than the one seek needed in the non-chunked SDS.

FIGURE 13m

Number of Seeks Needed to Access a Row of Data in a Chunked SDS

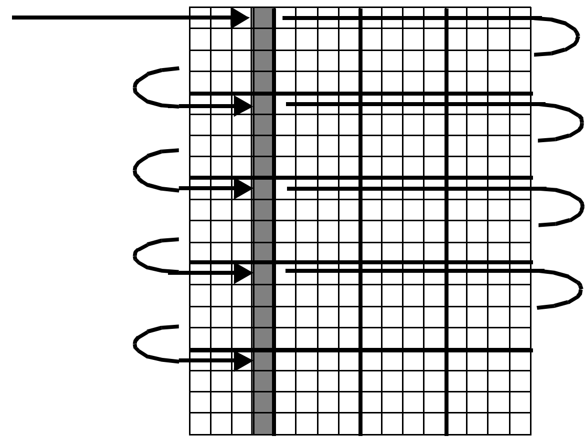
4 seeks are needed to find the starting location of a 1,400 integer row of data in a chunked data array with 400 *x* 400 integer chunks.



To read the aforementioned column of data, five chunks must be read into memory in order to access the 2,000 locations of the subset. Therefore, five seeks to the starting location of each of these chunks are necessary to complete the read operation - far fewer than the 2,000 needed in the non-chunked SDS.

FIGURE 13n **Number of Seeks Needed to Access a Column of Data in a Chunked SDS**

5 seeks are needed to find the starting location of a 2,000 integer column of data in a chunked data array with 400 *x* 400 integer chunks. (Each arrow represents one seek.)



These examples show that, in many cases, chunking can be used to reduce the I/O overhead of subsetting, but in certain cases, chunking can impair I/O performance.

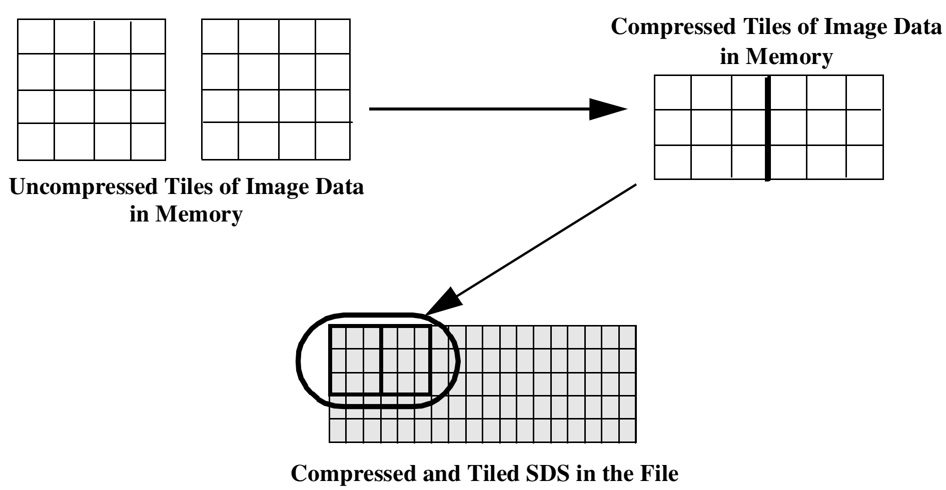
The efficiency of subsetting from chunked SDSs is partly determined by the size of the chunk - the smaller the chunk size, the more seeks will be necessary. Chunking can substantially improve I/O performance when data is read along the slowest-varying dimension. It can substantially degrade performance when data is read along the fastest-varying dimension.

13.3.4 Chunking with Compression

Chunking can be particularly effective when used in conjunction with compression. It allows subsets to be read (or written) without having to uncompress (or compress) the entire array.

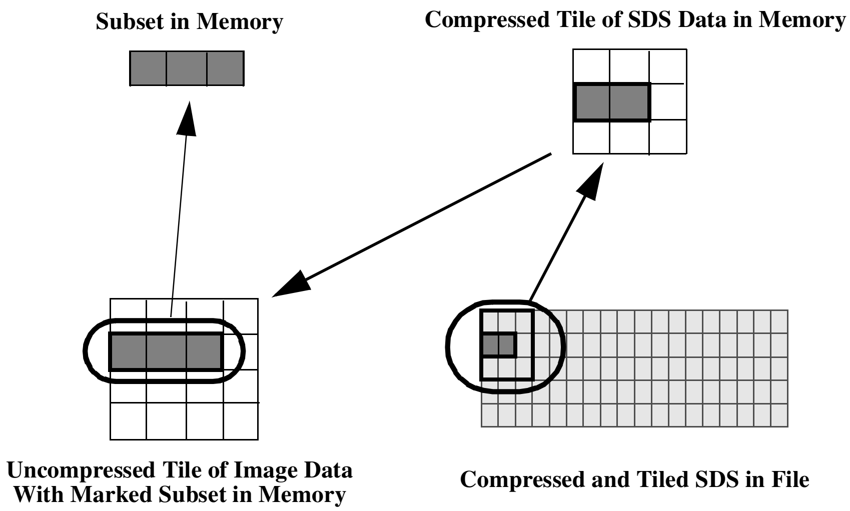
Consider the example of a tiled, two-dimensional SDS containing one million bytes of image data. Each tile of image data has been compressed as illustrated in the following figure.

FIGURE 13o **Compressing and Writing Chunks of Data to a Compressed and Tiled SDS**



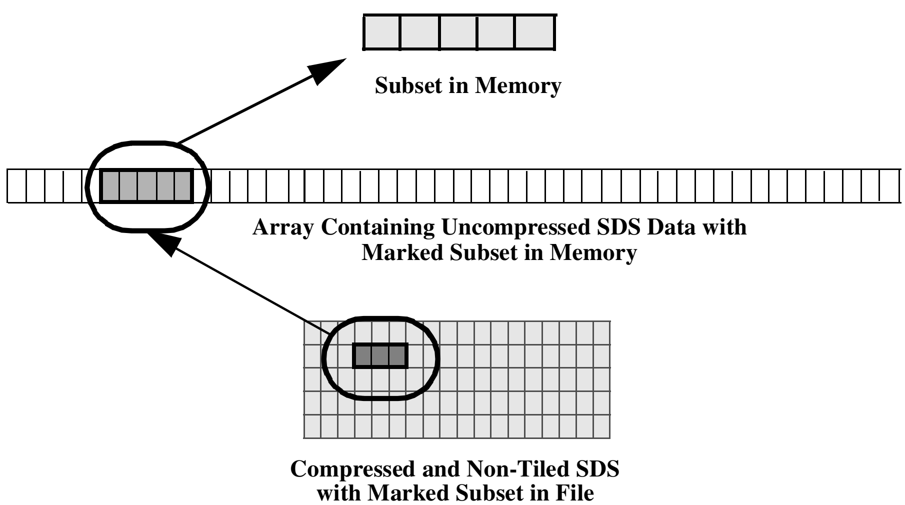
When it becomes necessary to read a subset of the image data, the application passes in the location of a tile, reads the entire tile into a buffer, and extracts the data-of-interest from that buffer.

FIGURE 13p **Extracting a Subset from a Compressed and Tiled SDS**



In a compressed and *non-tiled* SDS, retrieving a subset of the compressed image data necessitates reading the entire contents of the SDS array into a memory buffer and uncompressing it in-core. (See Figure 13p.) The subset is then extracted from this buffer. (Keep in mind that, even though the illustrations show two-dimensional data tiles for clarity, this process can be extended to data chunks of any number of dimensions.)

FIGURE 13q **Extracting a Subset from a Compressed Non-Tiled SDS**



As compressed image files can be as large as hundreds of megabytes in size, and a gigabyte or more uncompressed, it is clear that the I/O requirements of reading to and writing from non-tiled, compressed SDSs can be immense, if not prohibitive. Add to this the additional I/O burden inherent in situations where portions of several image files must be read at the same time for comparison, and the benefits of tiling become even more apparent.

NOTE: It's recommended that the **SDwritechunk** routine be used to write to a compressed and chunked SDS. **SDwritechunk** can perform this operation more efficiently than the combination of **SDsetcompress** and **SDwritedata**. This is because the chunk information provided by the user to the **SDwritechunk** routine must be retrieved from the file by **SDwritedata**, and therefore involves more computational overhead.

13.3.5 Effect of Chunk Size on Performance

The main concern in modelling data for chunking is that the chunk size be approximately equal to the average expected size of the data block needed by the application.

If the chunk size is substantially larger than this, increased I/O overhead will be involved in reading the chunk and increased performance overhead will be involved in the decompression of the data if it is compressed. If the chunk size is substantially smaller than this, increased performance and memory/disk storage overhead will be involved in the HDF library's operations of accessing and keeping track of more chunks, as well as the danger of exceeding the maximum number of chunks per file. (64K)

It is recommended that the chunk size be at least 8K bytes.

13.3.6 How Insufficient Chunk Cache Space can Impair Chunking Performance

The HDF library provides for the caching of chunks. This can substantially improve I/O performance when a particular chunk must be accessed more than once.

There is a potential performance problem when subsets are read from chunked datasets and insufficient chunk cache space has been allocated. The cause of this problem is the fact that two separate levels of the library are working to read the subset into memory and these two levels have a different "perspective" on how the data in the dataset is organized.

Specifically, higher-level routines like **SDreaddata** access the data in a strictly row-wise fashion - not according to the chunked layout. However, the lower-level code that directly performs the read operation accesses the data according to the chunked layout.

As an illustration of this, consider the 4 *x* 12 dataset depicted in the following figure.

FIGURE 13r **Example 4 *x* 12 Element Scientific Data Set**

1	2	3	4	5	6	7	8	9	10	11	12
21	22	23	24	25	26	27	28	29	30	31	32
41	42	43	44	45	46	47	48	49	50	51	52
61	62	63	64	65	66	67	68	69	70	71	72

Suppose this dataset is untiled, and the subset shown in the following figure must be read.

FIGURE 13s **2 *x* 8 Element Subset of the 4 *x* 12 Scientific Data Set**

1	2	3	4	5	6	7	8	9	10	11	12
21	22	23	24	25	26	27	28	29	30	31	32
41	42	43	44	45	46	47	48	49	50	51	52
61	62	63	64	65	66	67	68	69	70	71	72

As this dataset is untiled, the numbers are stored in linear order. **SDreaddata** finds the longest contiguous stream of numbers, and requests the lower level of the library code to read it into memory. First, the first row of numbers will be read:

```
3  4  5  6  7  8  9 10
```

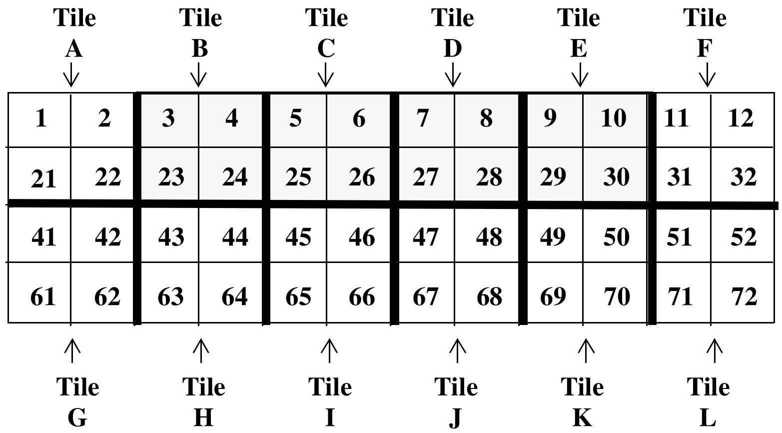
Then the second row:

```
23 24 25 26 27 28 29 30
```

This involves two reads, two disk accesses and sixteen numbers.

Now suppose that this dataset is tiled with 2 *x* 2 element tiles. On the disk, the data in this dataset is stored as twelve separate tiles, which for the purposes of this example will be labelled A through L.

FIGURE 13t **4 *x* 12 Element Data Set with 2 *x* 2 Element Tiles**



Also, the chunk cache size is set to 2.

A request is made to read the aforementioned subset of numbers into memory. As before, **SDreaddata** will determine the order the numbers will be read in. The routine has no information about the tiled layout. The higher-level code will again request the numbers "3, 4, 5, 6, 7, 8, 9, 10" from the lower level code on the first read operation.

In order to access those numbers the lower levels must read in four tiles: B, C, D, E. It reads in tiles B and C, retrieving the numbers "3, 4, 5, 6". However, as the cache space is now completely filled it must overwrite tile B in the cache to access the numbers "7, 8", which are in tile D. It then has to overwrite tile C to access "9, 10," which are in tile E. Note that, in each case, it does not use half of the numbers from the tile that it reads in, even though those numbers will be needed later.

Next, the higher-level code requests the second row of the subset - "23, 24, 25, 26, 27, 28, 29, 30." The lower-level code must *reread* tile B to access "23, 24", but tile B is no longer in the chunk cache. In order to access tile B, the lower-level code must overwrite tile D, and so on. By the time the subset read operation is complete, it has had to read in each of the tiles twice. Also, it has had to perform 8 disk accesses and has read 32 numbers.

Now consider a more practical example with the following parameters:

- A scientific data set has 3,000 rows and 8,400 columns.
- The target subset is 300 rows by 1,000 columns, and contains 300,000 numbers.

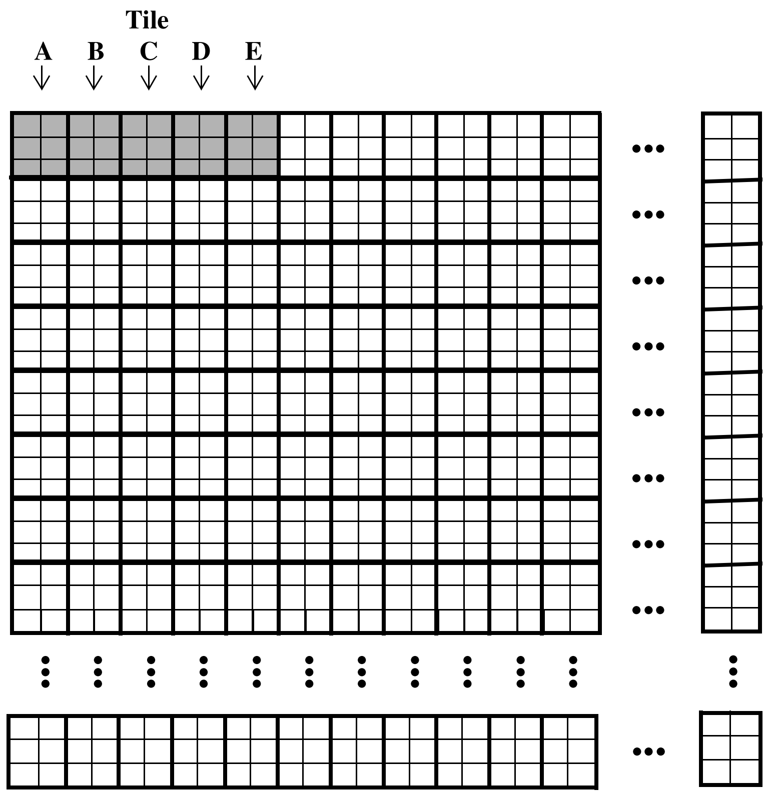
If the dataset is untiled the numbers are read into memory row-by-row. This involves 300 disk accesses for 300 rows, with each disk access reading in 1,000 numbers. The total number of numbers that will be read is 300,000.

Suppose the dataset is tiled as follows:

- The tile size is 300 rows by 200 columns, or 60,000 numbers.
- The size of the chunk cache is 2.

Each square in the following figure represents one 100 *x* 100 element region of the dataset. Five tiles span the 300 *x* 1,000 target subset. For the purposes of this example, they will be labelled A, B, C, D and E.

FIGURE 13u **5 200 *x* 300 Element Tiles Labelled A, B, C, D and E**



First, the higher-level code instructs the lower-level code to read in the first row of subset numbers. The lower-level code must read all five tiles (A through E) into memory, as they all contain numbers in the first row. Tiles A and B are read into the cache without problem, then the following set of cache overwrites occurs.

1. Tile A is overwritten when tile C is read.
2. Tile B is overwritten when tile D is read.
3. Tile C is overwritten when tile E is read.

When the first row has been read, the cache contains tiles D and E.

The second row is then read. The higher-level code first requests tile A, however the cache is full, so it must overwrite tile D to read tile A. Then the following set of cache overwrites occur.

1. Tile E is overwritten when tile B is read.
2. Tile A is overwritten when tile C is read.
3. Tile B is overwritten when tile D is read.
4. Tile C is overwritten when tile E is read.

For each row, five tiles must be read in. No actual caching results from this overwriting. When the subset read operation is complete, 300 * 5 = 1,500 tiles have been read, or 60,000 * 1,500 = 90,000,000 numbers.

Essentially, five times more disk accesses are being performed and 900 times more data is being read than with the untiled 3,000 *x* 8,400 dataset. The severity of the performance degradation increases in a non-linear fashion as the size of the dataset increases.

From this example it should be apparent that, to prevent this kind of chunk cache "thrashing" from occurring, the size of the chunk cache should be made equal to, or greater than, the number of chunks along the fastest-varying dimension of the dataset. In this case, the chunk cache size should be set to 4.

When a chunked SDS is opened for reading or writing, the default cache size is set to the number of chunks along the fastest-varying dimension of the SDS. This will prevent cache thrashing from occurring in situations where the user doesn't set the size of the the chunk cache. Caution should be exercised by the user when altering this default chunk cache size.

Chapter 14

HDF Command-Line Utilities

14.1 Chapter Overview

A number of command-line utilities are available for working with HDF files. They are described in this chapter.

14.2 The HDF Command-Line Utilities

The HDF command-line utilities are application programs that are executed from the UNIX shell prompt. These utilities serve the following needs of the HDF developer.

- They make it possible to perform, at the command line level, common operations on HDF files without having to resort to custom-programmed utilities to do these operations..
- They provide the capability for performing operations on HDF files that would be very difficult to do with custom-programmed utilities..

Table 14A lists the names and descriptions of the utilities described in this section. Descriptions and examples of these routines follow.

TABLE 14A **The HDF Command-Line Utilities**

Name	Description
hdfls	Displays the tags, reference numbers, and lengths of data elements.
hdfed	Displays the contents of an HDF file and allows limited manipulation of the data.
fp2hdf	Converts floating-point data to HDF floating-point format and/or HDF 8-bit raster image format (RIS8).
ristosds	Converts a set of RIS8 HDF files into a single three-dimensional SDS HDF file.
r8tohdf	Converts one or more 8-bit raster images in raw format to the HDF RIS8 format and writes them to a file, optionally with palettes.
hdftor8	Converts raster images and/or palettes from the HDF format to the raw format and stores them in two sets of files - one for images and the other for palettes.
hdfcomp	Compresses 8-bit raster images from an HDF file, storing them in a new HDF file.
r24hdf8	Converts raw RGB 24-bit images to an RIS8 with a palette.
paltohdf	Converts a raw palette to the HDF format.
hdftopal	Converts a palette in an HDF file to a raw palette format.
hdfpack	Compresses an HDF file, reading all of the objects in the file and writing them to a new HDF file.

Name	Description
vmake	Creates vsets.
vshow	Displays vset information.
hdp	Displays general information about the contents of an HDF file.

14.2.1 Listing Basic Information About an HDF File: hdfls

14.2.1.1 General Description

The `hdfls` utility provides general information about the tags, reference numbers, and if requested, lengths of the data elements.

14.2.1.2 Command-Line Syntax

```
hdfls [-o][-l][-d][-v][-g][-s][-h][-t #] filename
```

`-o`	Ordered	Indicates the reference numbers are to be displayed in ascending order.
`-l`	Long format	Displays more information about the file.
`-d`	Offset/ length	Offset and length information will be displayed for each element in the file.
`-v`	Verbose	Annotation and label text will be displayed, along with the format triggered by the -l flag. (long format)
`-g`	Group	List items by group.
`-s`	Special elements	Display detailed informaton about special elements.
`-h`	DD block	Dump DD block information.
`-t`	Tag	Only list information about the specified tag. Must be followed by a tag number.

14.2.1.3 Examples

A file called "aa.hdf" contains three items associated with a raster image; the image dimensions, a palette, and the raster image. To display information about the contents of this file, the following command is used.

```
hdfls aa.hdf
```

The following output is displayed:

```
aa.hdf:
Image Dimensions-8:      (Raster-8): (tag 200)
       Ref nos: 1
Image Palette-8:         (Raster-8): (tag 201)
       Ref nos: 3
Raster Image-8:          (Raster-8): (tag 202)
       Ref nos: 1
```

The following displays the same information with the length of each data element:

```
hdfls -l aa.hdf
```

This would result in the following information being displayed:

```
aa.hdf:
Image Dimensions-8:  (Raster-8) : (tag 200)
       Ref no 1                 4 bytes

Image Palette-8:     (Raster-8) : (tag 201)
       Ref no 3                 768 bytes

Raster Image-8:      (Raster-8) : (tag 202)
       Ref no 1                 120000 bytes
```

14.2.2 Editing the Contents of an HDF File: hdfed

14.2.2.1 General Description

The `hdfed` utility allows experienced HDF users to manipulate the elements of an HDF file. These manipulations include

- Selecting groups and showing information about them.
- Dumping group information to output files.
- Writing group data to output files.
- Deleting groups from HDF files.
- Inserting groups in HDF files.
- Replacing elements of HDF files.
- Editing the labels and descriptions of any element in an HDF file.

It is designed primarily for users who need to know about HDF files at the level of individual data elements. It is not designed to provide a comprehensive high-level view of the contents of an HDF file - other tools and utilities should be used for that purpose. To use `hdfed` one should be familiar with the components of an HDF file covered in the *HDF Specifications* manual.

The `hdfed` utility is loosely modeled on `ed`, the UNIX line editor. When `hdfed` is invoked, it prompts the user for commands, as does `ed`. Also, basic command syntax and description information is available to the user through `hdfed`. The most common `hdfed` commands are used to control the position in the HDF file and the format of the information provided.

The initial view of the file under `hdfed` consists of a set of tag/reference number pairs. Although `hdfed` allows modification of tags and reference numbers *within strict constraints,* it will not allow the user to arbitrarily modify binary data in the file.

The following terms and concepts must be understood in order to use `hdfed` correctly and will be used in the following discussion about `hdfed`.

- The ***data object*** or ***object*** refers to an HDF data object and the data descriptor of that object. (i.e., tags, reference numbers, offsets, or lengths.)
- The ***data*** or ***data element*** refers to the record that the data descriptor points to. For a precise definition of the data that is associated with a given tag consult the *HDF Specifications* manual.
- The ***group*** refers to a predefined collection of data objects that correspond to a particular application. For example, a raster image group refers to the collection of objects that are used to store all of the information in a raster image set.

Once an HDF file has been opened by `hdfed`, the following operations can be performed on the data file, among others:

- Select an HDF object to examine more closely.
- Move forward or backward within the HDF file.
- Get information about an object. (tag, reference number, size, label)
- Display a raster image using the ICR protocol.
- Display the contents of any object.
- Delete an object.
- Annotate an object with a label or description.
- Write an object to a second HDF file.
- Write data elements in binary form to a non-HDF file.
- Close the file and exit, or open a new file.

14.2.2.2 Command-Line Syntax

The syntax of `hdfed` is

```
hdfed [-nobackup][-batch] filename
```

If a file named `filename` exists, it is opened and a backup is made of the file. Files may also be opened from within the editor.

.

`-nobackup`	Specifies that no backup file is to be made. If this option is omitted, a backup file is automatically created.
`-batch`	Specifies that input to `hdfed` is to be input via a stream of `hdfed` commands, rather than interactively.

The `-batch` flag is useful when a group of commonly-used commands are included in a UNIX shell script. The following is an example of such a script, using the C-shell, that lists information about the groups in a specified HDF file.

```
#!/bin/csh -f
set file=$1
shift
hdfed -batch $file -nobackup << EOF
info -all group $*
close
quit
EOF
echo ""
```

To receive usage information, as well as a quick list of the `hdfed` commands, type the command

```
hdfed -help
```

While in `hdfed`, the standard command prompt is displayed.

```
hdfed>
```

Many `hdfed` commands have qualifiers, or flags. For example, the `info` command may be followed by the `-all`, `-long`, `-group`, or `-label` flags.

All of the commands and flags can be abbreviated to the extent that their abbreviations are unique. For example, `-he` is ambiguous as it could stand for either the `-hexadecimal` or the `-help` flags, but `-hel` is not ambiguous.

TABLE 14B **The `hdfed` Command Set**

Name	Description
help	Displays general `hdfed` help information.
open	Opens an HDF file.
close	Closes an HDF file.
revert	Reverts to the original HDF file.
next	Goes to the next object or group that satisfies the predicate.
prev	Goes to the previous object or group that satisfies the predicate.
info	Displays information about the current data object.
dump	Displays information about the current data object in non-default formats. (i.e., binary, ASCII, etc.). The default is octal.
display	Displays a raster image using ICR.
put	Writes the current data element in a non-HDF file with the specified filename in binary format.
putr8	Writes the current RIS8 group into a non-HDF file with the specified filename.
getr8	Reads a RIS8 group from a non-HDF file with the specified filename.
delete	Deletes an object or group.
write	Writes an object or group to an HDF file.
annotate	Annotates an object.
if	Conditional statement.
select	Loop for each object.
alias	Defines an alias or display the alias list.
unalias	Deletes an alias.
wait	Prints a message and wait for a carriage return.

To obtain information about the usage of any `hdfed` command, type the following at the `hdfed` prompt.

```
any hdfed command -help
```

Note that usage information cannot be obtained by typing only the command, with no flags. There are `hdfed` commands like `delete` that do not require an argument, so watch out for this kind of error.

There is a subset of `hdfed` commands where *predicates*, *items*, and *comparators* are used. ***Items*** are used to denote an HDF object type and can be any of the following identifiers; `tag`, `ref`, `image_size`, or `label`. A ***comparator*** is an expression used to compare an item with a user-defined value, and can be any of the following; "=" (equal to), "!=" (not equal to), ">" (less than), "<" (greater than), ">=" (greater than or equal to), "<=" (less than or equal to). User-defined values can be either a number (with or without a decimal point) or a string of characters delimited by double-quotes. ***Predicates*** consist of items, comparators and user-defined values and are of the syntax

```
item comparator value
```

Or they may consist of the identifier `group, as` in the `next group` command. Some examples of predicates are:

```
next group
next (same as "next group" as "group" is the default identifier)
next tag = 720
next ref = 2
next image_size < 1000
next label = "abc"
```

The following is a more inclusive description of the `hdfed` commands.

The `help` command

Syntax: **help**
Flags: None
Description: Prints a help screen describing the basic purpose and functionality of the hdfed utility.
Usage Example:

```
hdfed> help

hdfed allows sophisticated HDF users the ability to manipulate the
elements in an HDF file. These manipulations include selecting groups
...
```

The `open` command

Syntax: **open [-nobackup] filename**
Flags: **-nobackup** The specified file name is not backed up.
Description: Opens the specified HDF file.
Usage Example:

```
hdfed> open -help
open <file> [-nobackup]
-nobackup Don't make a backup for this file.
hdfed>
hdfed> open h1
hdfed>
```

The `info` command

Syntax: **info [-all] [-long] [-group] [-label]**
Flags:
-all Displays information for all of the objects in the current file.
-long Displays the long form of the information.
-group Organizes the information into groups.
-label Shows any labels.
Description: Displays information for a data object. The listing for special elements will contain a special tag value (in Item 13 below it's 18347, which corresponds to DFTAG_VS) and the text "Unknown Tag".
Usage Example:

```
hdfed> info -all -label -long
  (1)     Version Descriptor              : (Tag 30)
          Ref: 1, Offset: 202, Length     :92 (bytes)
  (2)     Scientific Data                 : (Tag 702)
          Ref: 2, Offset: 294, Length     : 200 (bytes)
  (3)     Number type                     : (Tag 106)
          Ref: 2, Offset: 494, Length     : 4 (bytes)
  (4)     SciData description             : (Tag 701)
          Ref: 2, Offset: 498, Length     : 2 (bytes)
  (5)     SciData max/min                 : (Tag 707)
          Ref: 2, Offset: 520, Length     : 4 (bytes)
```

```
        *(6)    Numeric Data Group              : (Tag 720)
                Ref: 2, Offset: 524, Length     : 12 (bytes)
                Label: Experiment #1
        (7)     Data Id Label                   : (Tag 104)
                Ref: 3, Offset: 536, Length     : 17 (bytes)
        (8)     Scientific Data                 : (Tag 702)
                Ref: 4, Offset: 553, Length     : 400 (bytes)
        (9)     Number type                     : (Tag 106)
                Ref: 4, Offset: 953, Length     : 4 (bytes)
        (10)    SciData description             : (Tag 701)
                Ref: 4, Offset:957, Length      : 22 (bytes)
        (11)    Numeric Data Group              : (Tag 720)
                Ref: 4, Offset: 979, Length     : 8 (bytes)
                Label: Experiment #2
        (12)    Data Id Label                   : (Tag 104)
                Ref: 5, Offset: 987, Length     : 17 (bytes)
        (13)    Unknown Tag                     : (Tag 18347)
                Ref: 8, Offset: 0, Length       : 40(bytes
hdfed>
hdfed> info -group -all
**Group 1:
                Numeric Data Group          : (Tag 720) Ref 2
                Scientific Data             : (Tag 702) Ref 2
                SciData description         : (Tag 701) Ref 2
                SciData max/min             : (Tag 707) Ref 2
**Group 2:
                Numeric Data Group          : (Tag 720) Ref 4
                Scientific Data             : (Tag 702) Ref 4
                SciData description         : (Tag 701) Ref 4
**These do not belong to any group:
                Version Descriptor          : (Tag 30) Ref 1
                Number Type                 : (Tag 106) Ref 2
                Data Id Label               : (Tag 104) Ref 3
                Number Type                 : (Tag 106) Ref 4
                Data Id Label               : (Tag 104) Ref 5
hdfed>
```

The `prev` command

Syntax: **prev <u>predicate list</u>**

Flags: None.

Description: Moves to the next object that satisfies the predicate list.

Usage Example:

```
hdfed> info -all
      (1)       Version Descriptor    : (Tag 30) Ref 1
      (2)       Scientific Data       : (Tag 702) Ref 2
      (3)       Number type           : (Tag 106) Ref 2
      (4)       SciData description   : (Tag 701) Ref 2
      (5)       SciData max/min       : (Tag 707) Ref 2
     *(6)       Numeric Data Group    : (Tag 720) Ref 2
      (7)       Data Id Label         : (Tag 104) Ref 3
      (8)       Scientific Data       : (Tag 702) Ref 4
      (9)       Number type           : (Tag 106) Ref 4
      (10)      SciData description   : (Tag 701) Ref 4
      (11)      Numeric Data Group    : (Tag 720) Ref 4
      (12)      Data Id Label         : (Tag 104) Ref 5
hdfed>
hdfed> ! The '*' in the first column marks the current
hdfed> ! position.
hdfed> ! The 'next' and 'prev' commands work with predicates.
hdfed> ! If I want to move to the max/min element,
hdfed> ! I can use the 'tag=' predicate.
hdfed>
hdfed> prev tag=707
hdfed> info
      (5)       SciData max/min (SciData) : (Tag 707) Ref:2
hdfed>
```

The `next` command

Syntax: **next** **predicate list**

Flags: None.

Description: Moves to the next object that satisfies the predicate.

Usage Example:

```
hdfed> ! Move in the file using next and prev
hdfed> ! The move direction depends on the relative positions.
hdfed> ! so it is often necessary to do an 'info -all' first.
hdfed> info -all
        (1)     Version Descriptor          : (Tag 30) Ref 1
        (2)     Scientific Data             : (Tag 702) Ref 2
        (3)     Number type                 : (Tag 106) Ref 2
        (4)     SciData description         : (Tag 701) Ref 2
       *(5)     SciData max/min             : (Tag 707) Ref 2
        (6)     Numeric Data Group          : (Tag 720) Ref 2
        (7)     Data Id Label               : (Tag 104) Ref 3
        (8)     Scientific Data             : (Tag 702) Ref 4
        (9)     Number type                 : (Tag 106) Ref 4
        (10)    SciData description         : (Tag 701) Ref 4
        (11)    Numeric Data Group          : (Tag 720) Ref 4
        (12)    Data Id Label               : (Tag 104) Ref 5
hdfed>
hdfed> ! This predicate persists for the next and prev
hdfed> ! commands. That means if I now type another 'next'
hdfed> ! command, it will look for a tag that equals 707.
hdfed>
hdfed> next
Reached end of file. Not moved.
hdfed> info
        (5)      SciData max.min (SciData)  : (Tag 707) Ref: 2
hdfed>
hdfed> next group
hdfed> next group
hdfed> info
        (11)    Numeric Data Group          : (Tag 720) Ref 4
hdfed>
```

The `dump` command

Syntax: **dump [-offset offset] [-length length] [-decimal|-short|-byte|-octal|-hexadecimal|-float|-double|-ascii]**

Flags:

-offset	Starting offset
-length	Length of the object to dump.
-decimal	Decimal format (32-bit integers)
-short	Decimal format (16-bit integers)
-byte	Decimal format (8-bit integers)
-octal	Octal format (the default)
-hexadecimal	Hexadecimal format
-float	Single-precision floating-point format (32-bit floats)
-double	Double-precision floating-point format (16-bit floats)
-ascii	ASCII format

Description: Displays the contents of the current object in the specified format.

Usage Example:

```
hdfed> ! to see the binary representation of this element
hdfed>
hdfed> dump
0: 257400004 257200004
hdfed>
hdfed> dump -short
hdfed>
0:      702     4       701     4
```

```
hdfed>
```

The `delete` command

Syntax:	**delete**
Flags:	None.
Description:	Deletes the current object or group.
Usage Example:	

```
hdfed> ! deleting groups
hdfed>
hdfed> ! If an element is required by other group it is alone.
hdfed> ! However, this is not perfect as the method by which group
hdfed> ! membership is determined can be pretty ad hoc.
hdfed>
hdfed> delete
hdfed> ! This deletes the Scientific Data Group
hdfed> info -all
      (1)      Version Descriptor       : (Tag 30) Ref 1
      (2)      Scientific Data          : (Tag 702) Ref 2
      (3)      Number type              : (Tag 106) Ref 2
      (4)      SciData description      : (Tag 701) Ref 2
      (5)      SciData max/min          : (Tag 707) Ref 2
      (6)      Numeric Data Group       : (Tag 720) Ref 2
      (7)      Data Id Label            : (Tag 104) Ref 3
      (8)      Number type              : (Tag 106) Ref 4
      (9)      Data Id Label            : (Tag 104) Ref 5
hdfed>
hdfed> ! Notice that the Numeric Data Group with reference
hdfed> ! number 4 is missing, and now there are only 9
hdfed> ! objects in the file.
hdfed>
```

The `annotate` command

Syntax:	**annotate [-label] [-descriptor] [-editor <u>editor</u>]**	
Flags:	**-label**	Edit a label (the default)
	-descriptor	Edit a descriptor.
	-editor	Use an editor. (Default is the editor referred to by the EDITOR environment variable.
Description:	Edits an annotation.	
Usage Example:		

```
hdfed>
hdfed> ! Annotations are labels and descriptors
hdfed>
hdfed> prev -group
hdfed> info -label
      (6)      Numeric Data Group    : (Tag 720) Ref 2
               Label: Experiment #1
hdfed> annotate -editor /usr/ucb/ex
"/tmp/he5091.1" 1 line, 14 characters
:p
Experiment #1
:s/$/      <more stuff>/
Experiment #1<more stuff>
:wq
"/tmp/he5091.1" 1 line 27 characters
hdfed> info -label
      (6)      Numeric Data Group    : (Tag 720) Ref 2
               Label: Experiment #1 <more stuff>
hdfed>
```

The `write` command

Syntax:	**write [-attachto <u>tag</u> <u>reference number</u>] <u>filename</u>**	
Flags:	**-attachto**	Which element the annotation will be attached to. (only for writing annotations)

Description: Writes an element or group into another HDF file.
Usage Example:

```
hdfed>
hdfed> ! Write object or group to another HDF file.
hdfed>
hdfed> write test
hdfed>
hdfed> ! Let's take a look at the file 'test'
hdfed> close; open test; info -all
      (1)      Version Descriptor    (Tag 30) Ref 1
      (2)      Scientific Data       (Tag 702) Ref 2
      (3)      Number type           (Tag 106) Ref 2
      (4)      SciData description   (Tag 701) Ref 2
      (5)      SciData max/min       (Tag 707) Ref 2
     *(6)      Numeric Data Group    (Tag 720) Ref 2
hdfed>
hdfed> close;
hdfed>
```

The `display` command

Syntax: **display [-position x-position y-position] [-expansion expansion] [-large]**

Flags:
- **-position** Image position on console screen
- **-expansion** Image expansion factor
- **-large** Make image as large as possible.

Description: Displays image on screen.
Usage Example:

```
hdfed> ! We will open a file with some RIS8 images.
hdfed>
hdfed> open denm,HDF
hdfed> display
hdfed>
hdfed> ! The 'display' command displays the current RIS8
hdfed> ! group image via ICR. I.e. if you are using NCSA Telnet
hdfed> ! on a Mac II, this would display the images from denm.HDF
hdfed> ! on your screen.
hdfed> ! NOTE: not guaranteed to work otherwise.
hdfed>
```

The `putr8` command

Syntax: **putr8 [-image image filename palette filename] -verbose]**

Flags:
- **-image** Image file name template (Default is "img#.@.%")
- **-palette** Palette file name template (Default is "pal#")
- **-verbose** To give output of steps taken.

Description: Writes a RIS8 group into raw image and palette files.
Usage Example:

```
hdfed> ! putr8 puts an RIS8 group into raw files
hdfed>
hdfed> putr8 -image my_image.#.@.% -palette testPalettes# -verbose
Writing to file: my_image8.10.10
Writing to file: my_palette
hdfed>
```

The `close` command

Syntax: **close [-keep]**

Flags:
- **-keep** The backup file is not deleted.

Description: Closes the HDF file opened by the last open command.
Usage Example:

```
hdfed> close
```

```
hdfed>
```

The `select` command

Syntax: **select predicate list command list**

Flags: None.

Description: Step through all the elements in the HDF file that satisfies the predicates, and execute the command list.

Usage Example:

```
hdfed> ! To step through a file and, for example, putr8 on all
hdfed> ! RIS8 groups we can use the select command.
hdfed>
hdfed> select tag=306
>> putr8 -image testImages# -palette testPalettes# -verbose
>> end
Writing to file: testImages8
Writing to file: testPalettes8
Writing to file: test Images14
Writing to file: testPalettes14
Writing to file: testImages21
Writing to file: testPalettes21
hdfed>
hdfed> ! The 'select' and 'if' commands take the same
hdfed> ! predicates as 'next' and 'pref'. There are also
hdfed> ! the predicates 'succeed" and "fail" that test the
hdfed> ! return status of the 'last' command.
hdfed>
```

The `put` command

Syntax: **put [-file filename] [-verbose]**

Flags:
- **-file** Output file name (Default is "elt#.@")
- **-verbose** Output diagnostic information.

Description: Writes the raw binary image of the current object to a file.

Usage Example:

```
hdfed> ! The 'put' command writes an element into a binary file.
hdfed> ! This is a dumb routine and does not know about the
hdfed> ! formats of an element.
hdfed>
hdfed> put -file binary#
hdfed> put -file myBinary -verbose
Writing to file: myBinary
hdfed>
```

The `revert` command

Syntax: **revert**

Flags: None.

Description: Discards all changes made in the current hdfed session.

Usage Example:

```
hdfed> revert
hdfed>
```

The `getr8` command

Syntax: **getr8 image file name [x-dimension y-dimension]**
[-palette palette file name]
[-raster|-rle|-imcomp]

Flags:
- **-palette** Palette will be read from a binary file.
- **-raster** No compression will be performed during the write. (the default)
- **-rle** Run-length compression will be performed during the write.
- **-imcomp** IMCOMP compression will be performed during the write.

```
Description:     Reads a RIS8 group from binary files.
```

The `if` conditional

```
Syntax:              if predicate list command list end
Flags:               None.
Description:         Executes commands in a loop if predicates are satisfied
                     for each element processed.
```

The `select` loop command

```
Syntax:              select predicate list command list end
Flags:               None.
Description:         Executes the list of commands for each element that
                     satisfies the predicates.
```

The `wait` command

```
Syntax:              wait message
Flags:               None.
Description:         Prints a message, then waits for a carriage return to be
                     typed.
```

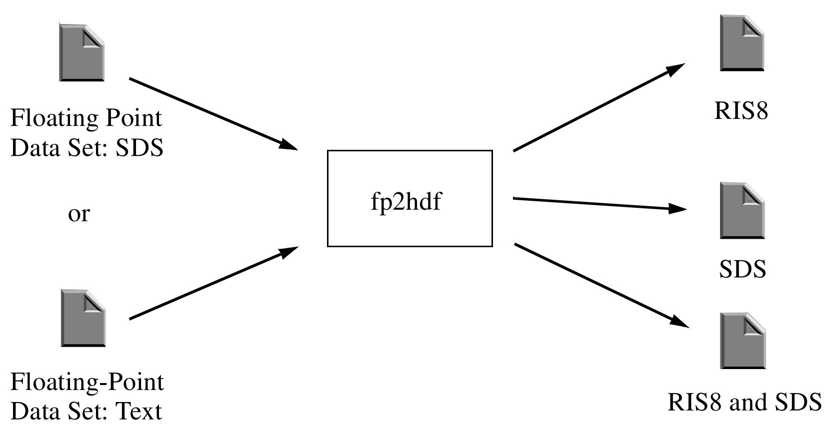
14.2.3 Converting Floating-Point Data to SDS or RIS8: fp2hdf

14.2.3.1 General Description

The `fp2hdf` utility converts 32-bit floating-point arrays from either text files or 32-bit HDF floating-point scientific data sets to either 8-bit HDF raster image sets or 32-bit floating-point HDF scientific data sets, or both, and stores the results in an HDF file. (See Figure 14a.) The images can be scaled on a user-specified mean value.

The term *scale* is used to refer to the area between points on the axes. If these gaps are of equal size, a uniform scale is specified - for example, "1.0, 2.0, 3.0, . . . ". In an HDF file, scales may be omitted, but in a text file (as in the text file shown below) they must be included.

FIGURE 14a **The fp2hdf Utility**



14.2.3.2 Command-Line Syntax

The syntax of `fp2hdf` is as follows.

```
fp2hdf input filename [-o output filename] [-r] [-f] [-e | -i]
   horizontal resolution vertical resolution [-p palette filename]
   [-m mean]
```

The input filename parameter is the name of the file containing the unconverted data set in HDF format. If the format is text, see the discussion on the following page about how it must be organized.

The output filename parameter is the name of the file containing the converted data set in HDF format. Depending on the output options output filename contains a scientific data set and/or raster image set for each of the data sets in the input files.

-e	Expand	Expands floating-point data using pixel replication (Default if -i option is specified)
-r	Raster	Stores the data as a raster image set in the output file.
-f	Float	Stores the data as a scientific data set in the output file. (Default if the "-r" option is not specified)
-i	Interpolation	Applies bilinear interpolation when expanding floating-point data. The horizontal resolution and vertical resolution parameters are, respectively, the horizontal and vertical resolution of the image
-p	Palette	Store the palette with the image. The palette filename parameter is the file name of the file containing the palette data.
-m	Mean	Cause the data to be scaled around the specified mean (supplied by the mean parameter) when generating the image, according to the following formulae: newmax = mean + 0.5*max(abs(max-mean), abs(mean-min) newmin = mean + 0.5*max(abs(max-mean), abs(mean-min))

The `-e` and the `-i` flags cannot be used simultaneously. Either pixel interpolation or bilinear interpolation can be chosen for image expansion, but not both.

If the `-i` option is chosen, the expanded image must have dimensions that are greater than or equal to the dimensions of the original data set.

An optional palette can accompany the image by loading it from an HDF file that contains a palette.

Data from several input files (with one set per input file) are stored as several data sets and/or images in one output file. A shell script can be used as another option in calling `fp2hdf` repeatedly to convert data from multiple input files to their corresponding output HDF files.

If an HDF file is used for input, it must contain an SDS. The SDS need only contain a dimension record and the data, but if it also contains maximum and minimum values and/or scales for the horizontal and vertical axes, these will be used as well.

If a text file is used for input, it must adhere to the following format.

```
number of rows number of columns
maximum value minimum value
scale for the vertical axis in an array
scale for the horizontal axis in an array
data element 1 data element 2 data element 3
...
```

The arrays that contain the scale for the vertical and horizontal axes must have a size equal to the values specified in the `number of rows` and `number of columns` positions, respectively. The data elements are floating-point data and are assumed to be ordered by rows, left-to-right and top-to-bottom.

14.2.3.3 Examples

This is the `fp2hdf` command-line syntax used to convert floating-point data in the file named "infile1.txt" to the SDS format, and to store it as an SDS in the HDF output file "outfile1".

```
fp2hdf infile1.txt -o outfile1
```

This `fp2hdf` command is used to convert floating-point data in the file named "infile2.txt" to an 8-bit raster image and store it in RIS8 format in the HDF output file named "outfile2".

```
fp2hdf infile2 -o outfile2 -r
```

This `fp2hdf` command is used to convert floating-point data in the file named "infile3.txt" to the RIS8 and SDS formats and store both converted data groups in the HDF output file "outfile3".

```
fp2hdf infile3.txt -o outfile3 -r -f
```

This `fp2hdf` command is used to convert floating-point data in the file named "infile4.txt" to a 500 x 600 raster image, storing the RIS8 in the HDF file "outfile4". This also stores the palette data read from the file name "palfile" with the image data.

```
fp2hdf input4.txt -o outfile4 -r -e 500 600 -p palfile
```

This `fp2hdf` command is used to convert floating-point data in all files whose names begin with the letter "f" to 500 x 600 RIS8 images and store them in the output file "output5".

```
fp2hdf f* -o outfile5 -r -i 500 600
```

14.2.4 Converting Several RIS8 Images to One 3D SDS: ristosds

14.2.4.1 General Description

The `ristosds` utility creates a single HDF file consisting of a three-dimensional SDS from a set of HDF files containing one or more raster images. All images in the input HDF files must have the same dimensions. If a palette is to be included with the images, it should be in the first HDF input file. Only one palette can be associated with the images; any additional palette data encountered by the utility after the first palette has been processed will be ignored.

14.2.4.2 Command-Line Syntax

```
ristosds input filename 1, input filename 2, ... input filename n
    [-o output filename]
```

14.2.4.3 Examples

The contents of a directory consists of 20 files named "storm001.hdf", "storm002.hdf". ... "storm020.hdf". Each file contains a single RIS8 with a 100 x 200 raster image. A file that combines these 20 raster images into a 32-bit floating-point SDS with the dimensions 100 x 200 x 20 can be created with the following `ristosds` command:

```
ristosds storm*.hdf -o storm.hdf
```

14.2.5 Converting 8-Bit Raster Images to the HDF Format: r8tohdf

14.2.5.1 General Description

The `r8tohdf` utility converts a set of raw raster images to the HDF RIS8 format and writes them to a file.

14.2.5.2 Command-Line Syntax

```
r8tohdf [number of rows number of columns] output filename
    [-p palette filename] [-c|-r|-i] raw raster image filename 1,
    raw raster image filename 2, ... raw raster image filename n
```

`-p`	Palette File	Inserts a palette stored in the file `palette filename` in the RIS8. If the -p flag is not specified, a palette is not stored with the RIS8.
`-c`	Run-length Encoding	Compresses the output data using run-length encoding.
`-i`	IMCOMP Compression	Compresses the output data using the IMCOMP method.
`-r`	No Compression	No compression is applied to the output data. (the default)

14.2.5.3 Examples

A file named "rawras" contains a 256 x 512-byte raw raster image, and its palette is stored in a file name "mypal". To convert the information in these files to an RIS8 without compression and store the RIS8 in a file named "ras.hdf", enter the following `r8tohdf` command:

```
r8tohdf 256 512 ras.hdf -p mypal rawras
```

A 800 x 1000-byte raw raster image is stored in a file named "bigpic". This data must be converted to a RIS8 without a palette, compressing it using run-length encoding, then stored in a file named "bigpic.hdf". The following command will do this:

```
r8tohdf 800 1000 bigpic.hdf -c bigpic
```

A 300 x 400 raw raster image is contained in each of the files named "pic1", "pic2", and "pic3". To convert all three files to RIS8s, compress them using the IMCOMP method, and store them in a file named "pic.hdf", enter

```
r8tohdf 300 400 pic.hdf -i pic1 pic2 pic3
```

Different types of raster image data are to be stored in a file named "ras.hdf". The image data in the file "rawras1" will be stored without a palette. The image data sets from the file named "rawras2" are to be stored with a palette extracted from a file named "mypal". The images from the "rawras1" and "rawras2" files are to be compressed using run-length encoding, and the image in the "rawras3" file is not to be compressed. The size of all images are 256 x 512 bytes. The following command is used to do this:

```
r8tohdf 256 512 ras.hdf -c rawras1 -p mypal rawras2 -r rawras3
```

14.2.6 Extracting 8-Bit Raster Images and Palettes from HDF Files: hdftor8

14.2.6.1 General Description

The `hdftor8` utility extracts the raster images and/or palettes from an HDF file and stores them in one file that contains the raster image data and another that contains the palette data.

14.2.6.2 Command-Line Syntax

```
hdftor8 input HDF filename [-i] [-v] [-r raster image filename]
    [-p palette filename]
```

-i	Interactive Mode	Program is executed in interactive mode.
-v	Verbose Mode	Program is executed in verbose mode. Diagnostic messages are displayed during the session.
-r	Raster Image File Name	The raster image file name immediately follows this flag.
-p	Palette File Name	The palette file name immediately follows this flag.

The names given as the HDF format file, raster image file, and palette file are interpreted by `hdftor8` as follows: For each raster image file, the file name is given the extension

```
.#.@.%
```

where "#" represents the raster image number from the HDF file, "@" represents the x-dimension of the raster image and "%" represents the y-dimension of the raster image. For each palette file, the file name is given the extensions ".#", where "#" represents the palette number from the HDF format file.

If no name is given for the raster image file, the default name "img.#.@.%" is assigned, where "#", "@", and "%" are defined as in the preceding paragraph. The default name for a palette file, if no name is specifically given in the command, is "pal.#".

14.2.6.3 Examples

A file named "denm.hdf" contains three 512 x 256 raster images and three palettes. To store these images and palettes in separate raster image and palette files, use the following `hdftor8` command:

```
hdftor8 denm.hdf
```

Six files are created, named "img1.512.256:", "img2.512.256", "img3.512.256", "pal.1", "pal.2", and "pal.3".

14.2.7 Compressing RIS8 Images in an HDF File: hdfcomp

14.2.7.1 General Description

The `hdfcomp` utility reads RIS8 images from a set of HDF files, compresses them and stores the compressed data in a second HDF file. If the output HDF file exists, the compressed images will be appended to it.

14.2.7.2 Command-Line Syntax

```
hdfcomp output_filename [-c|-r|-i] input_filename_1,
   [-c|-r|-i] input_filename_2, ... [-c|-r|-i] input_filename_n
```

`-r`	No compression	The raster image data is not compressed. (the default)
`-c`	Run-length Encoding	The raster image data is compressed using run-length encoding.
`-i`	IMCOMP Compression	The raster image data is compressed using the IMCOMP algorithm.

14.2.7.3 Examples

A directory contains twenty files named "storm001", "storm002", ... "storm020". Each of these files contains a single RIS8 image. To compress these images using run-length encoding and store them in a file named"altcomp.hdf", use the following `hdfcomp` command:

```
hdfcomp allcomp.hdf -c storm*.hdf
```

14.2.8 Converting 24-Bit Raw Raster Images to RIS8 Images: r24hdf8

14.2.8.1 General Description

The `r24hdf8` utility quantizes a raw RGB 24-bit raster image, creating an 8-bit image with a 256-color palette, then it stores the palette and raster image data in an HDF file.

14.2.8.2 Command-Line Syntax

```
r24hdf8 [x-dimension length y-dimension length] raw 24-bit image file-
    name hdf ris8 image filename
```

The pixel order in the raw 24-bit image file is left-to-right and top-to-bottom. Each pixel data element consists of three contiguous bytes, the first representing the red intensity value, the second the green intensity value, and the third the blue intensity value. Use the `ptox` filter to convert the raster image data from a pixel-interlaced format to scan-plane interlaced.

14.2.8.3 Examples

A file named "rawraster" containing 24-bit raw raster images with x and y-dimensions of 480 x 640, respectively, must be converted to the HDF RIS8 format and stored in a file named "hdfraster". The following command is used to do this:

```
r24hdf8 480 640 rawraster hdfraster
```

14.2.9 Converting an HDF RIS24 Image to an HDF RIS8 Image: hdf24hdf8

14.2.9.1 General Description

The `hdf24hdf8` utility quantizes an HDF RGB RIS24 pixel-interlaced image, producing an HDF RIS8 image with a 256-color palette and stores the palette and raster image data in an HDF file.

14.2.9.2 Command-Line Syntax

```
hdf24hdf8 ris24 image filename ris8 image filename
```

14.2.10 Converting Raw Palette Data to the HDF Palette Format: paltohdf

14.2.10.1 General Description

The `paltohdf` utility converts raw palette data to the HDF palette format. The raw palette data must have 768 bytes organized in the following order: first, 256 contiguous red intensity values, then 256 contiguous green intensity values, then 256 contiguous blue intensity values. The palette in the HDF file will have the RGB values pixel-interlaced, as follows.

```
red value green value blue value red value green value
    blue value ...
```

This is the standard HDF format for 8-bit palettes.

14.2.10.2 Command-Line Syntax

```
paltohdf raw format palette filename HDF format palette filename
```

If a HDF palette format file is specified that doesn't exist, it is created before the converted data is stored. If an HDF palette format file is specified that already exists, the converted data is appended to the file.

14.2.11 Extracting Palette Data from an HDF File: hdftopal

14.2.11.1 General Description

The `hdftopal` utility converts a palette in an HDF file to a raw palette in an non-HDF file. The raw palette will have 768 bytes with the first 256 bytes representing red intensity values, the second 256 bytes representing green intensity values, and the third 256 bytes representing blue intensity values. The utility performs the converse operation of the `paltohdf` utility.

14.2.11.2 Command-Line Syntax

```
hdftopal HDF format palette filename raw format palette filename
```

14.2.12 Compressing an HDF File: hdfpack

14.2.12.1 General Description

The `hdfpack` utility compresses all of the data in an HDF file and writes the compressed data to a second HDF file.

14.2.12.2 Command-Line Syntax

```
hdfpack [-i|-b] [-d number of data descriptors per block]
    [-t number of linked blocks per table entry] input HDF filename
    output HDF filename
```

`-b`	Non-coalesced Blocks	The utility will not coalesce linked-block elements.
`-i`	Interactive Mode	The utility will prompt for each linked-block element.
`-d`	Data descriptors per block	The output file will be created with the specified number of data descriptors per block of data descriptors.
`-t`	Linked-blocks per table entry	The output file will be created with the specified number of linked blocks per table entry.

14.2.12.3 Examples

To compress the data in the file named "aa.hdf" and store the compressed data in the file named "aa.cmp", use the following `hdfpack` command:

```
hdfpack aa.hdf aa.cmp
```

Suppose a file named "bb.hdf" contains data elements stored as sequences of linked blocks. The following `hdfpack` command compresses the file while leaving the linked-block elements intact, and writes the compressed data to a file named "bb.blk".

```
hdfpack -b bb.hdf bb.blk
```

14.2.13 Displaying Vdata Information: vshow

14.2.13.1 General Description

Displays information about either one Vdata object in an HDF file, or all Vdata objects in the file.

14.2.13.2 Command-Line Syntax

```
vshow input HDF filename [+|+vdata_id]
```

+	All Vdatas	The utility will display information about all Vdata objects in the HDF file.
+vdata_id	One Vdata	The utility will display information about the Vdata object corresponding to the specified vdata id.

14.2.13.3 Examples

Information about all of the Vdata objects in the HDF file named "image012.hdf" must be examined. The following command will display this information.

```
vshow image012.hdf +
```

14.2.14 Displaying General Information About the Contents of an HDF File: hdp

14.2.14.1 General Description

The `hdp` utility provides quick and general information about all objects in the specified HDF file. It can list the contents of HDF files at various levels with different details. It can also dump the data of one or more specific objects in the file.

14.2.14.2 Command-Line Syntax

```
hdp [-H command] filename
```

-H	Help	Displays usage information about the specified command. If no command is listed, information about all commands are displayed.

Like `hdfed`, hdp provides a set of commands that allow the user to determine what kind of information is to be displayed.

TABLE 14C **The `hdp` Command Set**

Name	Description
list	Displays the contents of the HDF files in the specified format.
dumpsds	Displays the contents of the SDSs in the listed files.
dumpvd	Displays the contents of the vdata objects in the listed files.
dumpvg	Displays the contents of the vgroup objects in the listed files.
dumprig	Displays the contents of the RIGs in the listed files.

The `list` command

Syntax: **list [-s|-l|-d] [-n|-c|-a] [-g|-t number name] [-ot|-of|-og|-on] message**

Flags:

-s	Short format.
-l	Long format.
-d	Debug format.
-n	Display the object name.
-c	Display the object class.
-a	Display the object description.
-g	Display groups only.
-t	Display objects with the specified tag number or name.
-ot	Sort by tag.
-of	Sort by position in the data descriptor list.
-og	Sort by group.
-on	Sort by name.

Description: Display the contents of the HDF files in the specified format. As with the 'info' command, the listing for specialelements will contain a special tag value (for DFTAG_VS it's 18347) and the text "Unknown Tag".

The `dumpsds` command

Syntax: **dumpsds [-i index | -r ref list | -n name list | -a] [-v|-h|-d] [-o filename | -b | -t]**

Flags:

-i	Dump SDSs with the specified index.
-r	Dump SDSs with the specified reference numbers.
-n	Dump SDSs with the specified names.
-a	Dump all SDSs.
-v	Display all SDS contents including annotations.
-h	Display the SDS header only.
-d	Display the SDS data only.
-o	Print information to the specified file.
-b	Specify output file as binary.
-t	Specify output file as text.

Description: Displays SDS information in the specified format.

The `dumpvd` command

Syntax: **dumpvd [-i index | -r ref list | -n name list | -c class | [-a] [-v|-h|-f|-d] [-o filename | -b | -t]**

Flags:

-i	Dump vdatas with the specified index.
-r	Dump vdatas with the specified reference numbers.
-c	Dump vdatas with the specified class.
-n	Dump vdatas with the specified names.
-a	Dump all vdatas.
-v	Display all vdata contents including annotations.
-h	Display the vdata header only.
-d	Display the vdata data only.
-f	Display the vdata field data only.

-o	Print information to the specified file.
-b	Specify output file as binary.
-t	Specify output file as text.

Description: Displays vdata information in the specified format.

The dumpvg command

Syntax: **dumpvg [-i index | -r ref list | -n name list | -c class | [-a] [-v|-h|-d] [-o filename | -b | -t]**

Flags:

-i	Dump vgroups with the specified index.
-r	Dump vgroups with the specified reference numbers.
-c	Dump vgroups with the specified class.
-n	Dump vgroups with the specified names.
-a	Dump all vgroups.
-v	Display all vgroup contents including annotations.
-h	Display the vgroup header only.
-d	Display the vgroup data only.
-o	Print information to the specified file.
-b	Specify output file as binary.
-t	Specify output file as text.

Description: Displays vgroup information in the specified format.

The dumprig command

Syntax: **dumprig [-i index | -r ref list | -m <8, 24> | [-a] [-v|-h|-d] [-o filename | -b | -t]**

Flags:

-i	Dump RIGs with the specified index.
-r	Dump RIGs with the specified reference numbers.
-m	Dump RIGs with the specified data length - 8- or 24-bit.
-a	Dump all RIGs.
-v	Display RIG contents including annotations.
-h	Display the RIG header only.
-d	Display the RIG data only.
-o	Print information to the specified file.
-b	Specify output file as binary.
-t	Specify output file as text.

Description: Displays RIG information in the specified format.

14.2.15 The HDF User-Contributed Utilities

In addition to the command-line utilities supported by NCSA, a number of utilities have been contributed by HDF users. Although they are not supported by NCSA, these utilities are distributed by NCSA via anonymous ftp at hdf.ncsa.uiuc.edu in the "/pub/dist/HDF/contrib" directory. These utilities, which are listed and briefly described in Table 14D, will not be discussed in this manual.

Note that this list is current as of the release date of this manual. A updated list of all of the user-contributed HDF utilities is available in the "README" file at the HDF ftp site and directory location mentioned above.

TABLE 14D

HDF User-Contributed Utilities

Name	Description
yuvconvert	Converts raster images in an HDF file to mpeg format.
LinkWinds	Interactively accesses, displays and performs analysis on multidisciplinary data sets. Includes its own graphical user interface.
fixver	Removes garbage data from the end of the version tag in the specified file for the benefit of applications, like the Macintosh port of NCSA Collage, that do not deal with this garbage data well.
ReadDF	Reads the contents of an HDF or a netCDF file into the Silicon Graphics Iris Explorer.
hdfinfo	Displays general information about the contents of an HDF file.
hdf24seq	Sequentially displays 24-bit raster images in HDF files. Tested on IBM RS/6000 and some SGI workstations.
mhdf24seq	Provides the same functionality as `hdf24seq`, but allows the user to display raster image data from more than one HDF file at a time.
vs2ris	Converts Vset data in the specified file to HDF RIS format.
vs2ps	Converts Vset data in the specified file to PostScript format.
inspectHDF	Displays general information about the contents of an HDF file.
fits2hdf	Converts FITS-formatted data to HDF format.
fix32luf	Fixes a bug in HDF version 3.2 revisions 1, 2 and 3 where SDS strings are written incorrectly.
sds2ris	Converts SDS data in the specified input file to RIS format, then writes the converted data to the specified output file.
cgmct.to.raster.npoc	Converts graphical data from a CGM clear text metafile, as outputted by CA-DISSPLA and CA-GKS to either raw raster images and CLUTs and stores it in a generic output file, or to RIS8 data and stores it in an HDF file.
cgmct.to.raster.withpc	Converts graphical data from a CGM clear text metafile, as outputted by CA-DISSPLA and CA-GKS to either raw raster images and CLUTs and stores it in a generic output file, or to RIS8 data and stores it in an HDF file, or a NERSC compressed movie file.
hdfrseq_new	New `hdfrseq` routine with SunView features.
hdftodico	Converts an HDF-formatted file to a Dicomed DCN file.
hdftops	Converts HDF raster image data to PostScript format.
iristohdf	Converts data in the Silicon Graphics image format to the HDF format.
isistoddc	Converts data in the Silicon Graphics image format to the Dicomed D48 DDC format.
xwdtohdf	Converts data in the X-Windows display format to the HDF RIS format.
hdftoxwd	Converts data in the HDF RIS format to the X-Windows display format.
qdvtohdf	Converts data in the qdv format to the HDF RIS format.
paltohdf	Converts raw palette data to the HDF palette format.
hdfxdis	Displays HDF RIS data on an X-Windows server.
hdf2tiff	Converts HDF RIS data to the TIFF image format.
tiff2hdf	Converts data in the TIFF image format to the HDF RIS format.
hdf2ras	Converts HDF RIS data to the Sun Rasterfile format.
ras2hdf	Converts raster image data in the Sun Rasterfile format to the HDF RIS format.
vs2ps	Converts polygonal Vset data to contour plots in PostScript format.
vs2ris	Converts polygonal Vset data to HDF RIS format.

Appendices

Appendix A: NCSA HDF Tags

Overview

This appendix includes tables containing brief descriptions of most of the tags that have been assigned at NCSA for general use. This list will be expanded in future editions to include new tags as they are assigned. A more detailed description of the tags can be found in the *HDF Specifications* manual.

Each table contains a list of tags within one category. The titles of the tables, with a functional description of each table, are:

- **Table A: The HDF Utility Tags.** Used by the HDF utilties.
- **Table B: The HDF General Raster Image Tags.** Used to describe aspects of raster image data.
- **Table C: The HDF Composite Image Tags.** Used to describe aspects of composite image data.
- **Table D: The HDF Scientific Data Set Tags:** Used to describe aspects of scientific data set (SDS) data.
- **Table E: The HDF Vset Tags.** Used to describe aspects of HDF Vset data.
- **Table F: The Obsolete HDF Tags:** Used to describe aspects of HDF data elements that have been replaced by newer tags or discontinued.

Tag Types and Descriptions

The following tables have five columns: the "Tag Name" column contains the abbreviated symbolic names of tags that are often used in an augmented form in HDF programs, the "Short Description" column contains a brief (four word maximum) description of the tag that is commonly used to describe to the tag in HDF manuals and in-line code documentation, the "Data Size" column describes the type of data that is associated with the tag and, where possible, lists the data size, the "Number" column list the numeric value of the tag symbol in the "hdf.h" header file, and the "Long Description" column contains a general description of the tag. The "Data Type" column contains a generalized description of the type of data referred to by the tag.

In the tables, any entry specified as "String" refers to a sequence of ASCII characters with the null byte possibly occurring at the end, but nowhere else. Any entry specified as "Text" also refers to a sequence of ASCII characters, but it may contain zero bytes anywhere in the sequence. An entry in the "Data Type" specified by an "n" refers to a data unit of variable-length. For a more detailed description of these units of data, refer to the appropriate tag entry in the *HDF Specification Manual*.

TABLE A

The HDF Utility Tags

Tag Name	Short Description	Data Size	Tag Value	Long Description
DFTAG_NULL	No Data	None	001	Used for place holding and filling up empty portions of the Data Descriptor Block.
DFTAG_VERSION	Library Version Number	4 bytes + string	030	Specifies the latest version of the HDF library used to write to the file.
DFTAG_NT	Number Type	4 bytes	106	Used by any other element in the file to specifically indicate what a numeric value looks like.
DFTAG_MT	Machine Type	0 bytes	107	Specifies that all unconstrained or partially constrained values in this HDF file are of the default type for that hardware.
DFTAG_FID	File Identifier	String	100	Points to a string that the user wants to associate with this file. This supports the inclusion of a user-supplied title for the file.
DFTAG_FD	File Descriptor	Text	101	Points to a block of text describing the overall file contents. It is intended to be user-supplied comments about the file.
DFTAG_TID	Tag Identifier	String	102	Provides a way to determine the meaning of a tag stored in the file.
DFTAG_TD	Tag Descriptor	Text	103	Similar to DFTAG_TD, but allows more text to be included.
DFTAG_DIL	Data Identifier Label	String	104	Associates the string with the Data Identifier as a label for whatever the identifier points to. By including DILs, any data element can be given a label for future reference. For example, this tag is often used to give titles to raster image data sets.
DFTAG_DIA	Data Identifier Annotation	Text	105	Associates the text block with the Data Identifier as an annotation for whatever that Data Identifier points to. With DIAs, and Data Identifier can have a lengthy, user-provided description of why that particular data element is in the file.
DFTAG_RLE	Run-length Encoding	0 bytes	011	Specifies that run-length encoding (RLE) is used to compress a raster image.
DFTAG_IMC	IMCOMP Compression	0 bytes	012	Specifies that IMCOMP compression is used to compress a raster image.
DFTAG_JPEG	24-bit JPEG Compression	n bytes	013	Provides header information for 24-bit JPEG-compressed raster images.
DFTAG_GREYPEG	8-bit JPEG Compression	n bytes	014	Provides header information for 8-bit JPEG-compressed raster images.

TABLE B

The HDF General Raster Image Tags

Tag Name	Short Description	Data Size	Tag Value	Long Description
DFTAG_RIG	Raster Image Group	n*4 bytes	306	Lists the Data Identifiers (tag/reference number pairs) that uniquely describe a raster image set.
DFTAG_ID	Image Dimension	20 bytes	300	Defines the dimensions of the two-dimensional array the corresponding RI tag refers to.
DFTAG_LD	LUT Dimension	20 bytes	307	Defines the dimensions of the two-dimensional array the corresponding LUT tag refers to.
DFTAG_MD	Matte Dimension	20 bytes	308	Defines the dimensions of the two-dimensional array the corresponding MA tag refers to.
DFTAG_RI	Raster Image	x*y bytes	302	Points to a raster image data set.
DFTAG_CI	Compressed Image	n bytes	303	Points to a compressed raster image data set.
DFTAG_LUT	Lookup Table	n bytes	301	Table to be used by the hardware for the purpose of assigning RGB or HSV colors to data values.
DFTAG_MA	Matte Data	n bytes	309	Points to matte data.
DFTAG_CCN	Color Correction	n bytes	310	Specifies the gamma correction for the raster image and color primaries used in the generation of the image.
DFTAG_CFM	Color Format	String	311	Indicates the interpretation to be given to each element of each pixel in a raster image.
DFTAG_AR	Aspect Ratio	4 bytes	312	Indicates the aspect ratio of the image.
DFTAG_XYP	XY Position	8 bytes	500	Specifies the screen X-Y coordinate for raster image sets. (Also used for composite image sets - See the entry for DFTAG_XYP in Table 12.6)

TABLE C

The HDF Composite Image Tags

Tag Name	Short Description	Data Size	Tag Value	Long Description
DFTAG_DRAW	Draw	n*4 bytes	400	Specifies a list of Data Identifiers (tag/reference number pairs) which define a composite image.
DFTAG_XYP	XY Position	8 bytes	500	Specifies the screen X-Y coordinate for composite image sets. (Also used for raster image sets - See the entry for DFTAG_XYP in Table 12.5)
DFTAG_RUN	Run	n bytes	401	Identifies code that is to be executes as a program or script.
DFTAG_T14	Tektronix 4014	n bytes	602	**Used as a vector image tag.** Points to a Tektronix 4014 data. The bytes in the data field, when read and sent to a Tektronix 4014 terminal, will be displayed as a vector image.
DFTAG_T10S	Tektronix 4015	n bytes	603	**Used as a vector image tag.** Points to a Tektronix 4015 data. The bytes in the data field, when read and sent to a Tektronix 4015 terminal, will be displayed as a vector image.

TABLE D

The HDF Scientific Data Set Tags

Tag Name	Short Description	Data Size	Tag Value	Long Description
DFTAG_NDG	Numeric Data Group	n*4 bytes	720	Lists the Data Identifiers (tag/reference number pairs) that describe a scientific dataset. Supersedes DFTAG_SDG.
DFTAG_SDD	SDS Dimension Record	n bytes	701	Defines the rank and dimensions of the array the corresponding SD refers to.
DFTAG_SD	Scientific Data	Real Number	702	Points to scientific data.
DFTAG_SDS	SCales	Real Number	703	Identifies the scales to be used when interpreting and displaying data.
DFTAG_SDL	Labels	String	704	Labels all dimensions and data.
DFTAG_SDU	Units	String	705	Displays units for all dimensions and data.
DFTAG_SDF	Formats	String	706	Displays formats for axes and data.
DFTAG_SDM	Maximum/minimum	2 Real Numbers	707	Displays the maximum and minimum values for the data.
DFTAG_SDC	Coordinate system	String	708	Displays the coordinate system to be used in interpreting data.
DFTAG_SDLNK	SDS Link	8 bytes	710	Links and old-style DFTAG_SDG and a DFTAG_NDG in cases where the DFTAG_NDG meets all criteria for a DFTAG_SDG.
DFTAG_CAL	Calibration Information	36 bytes	731	The calibration record for the corresponding DFTAG.SD.
DFTAG_FV	Fill Value	n bytes	732	The value which has been used to indicate unset values in the corresponding DFTAG_SD.

TABLE E

The HDF Vset Tags

Tag Name	Short Description	Data Size	Tag Value	Long Description
DFTAG_VG	Vgroup	14+n bytes	1965	Provides a general-purpose grouping structure.
DFTAG_VH	Vdata Description	22+n bytes	1962	Provides information necessary to process a DFTAG_VS.
DFTAG_VS	Vdata	n bytes	1963	Contains a block a data that is to be interpreted according to the information in the corresponding DFTAG_VH.

TABLE F

The Obsolete HDF Tags

Tag Name	Short Description	Data Size	Tag Value	Long Description
DFTAG_IDS	Image Dimension-8	4 bytes	200	Two 16-bit integers that represent the width and height of an 8-bit raster image in bytes.
DFTAG_IP8	Image Palette-8	768 bytes	201	A 256 x 3 byte array representing the red, green and blue elements of the 256-color palette respectively.
DFTAG_RI8	Raster Image-8	x*y bytes	202	A row-oriented representation of the elementary 8-bit image data.

Tag Name	Short Description	Data Size	Tag Value	Long Description
DFTAG_CI8	Compressed Image-8	n bytes	203	A row-oriented representation of the elementary 8-bit raster image data, with each row compressed using a form of run-length encoding.
DFTAG_II8	IMCOMP Image-8	n bytes	204	A 4:1 8-bit raster image, compressed using the IMCOMP algorithm.
DFTAG_SDG	Scientific Data Group	n*4 bytes	700	List the Data Identifiers (tag/reference number pairs) that uniquely describe a scientific dataset.
DFTAG_SDT	Transpose	0 bytes	709	Indicates that data is transposed in the file.

Appendix B: HDF Installation Overview

General HDF Installation Overview

A. Acquiring the HDF Library Source.

You may obtain HDF via FTP, an archive server, or US mail.

FTP server: If you are connected to Internet (NSFNET, ARPANET, MILNET, etc.) you may download HDF source code at no charge from the anonymous ftp server at NCSA. The Internet address of the server is:

```
ftp.ncsa.uiuc.edu or 141.142.3.135
```

Note: the IP address has been changed from 141.142.20.50 to 141.142.3.135. Log in by entering anonymous for the name and your local e-mail address ("login@host") for the password.

After logging in change directory to "HDF/HDF4.1/". If you want packed source code, change directory to "tar/", "hqx/", or "zip/". Files in those directories must be transferred using binary mode.

If you want unpacked source code, change directory to "unpacked/" and transfer all the files in unpacked/ and in its subdirectories to your host.

If you have any questions regarding this procedure or whether you are connected to Internet, consult your local system administration or network expert.

We have set up an HDF anonymous FTP server to mirror what is on the FTP server. The internet address of the HDF server is:

```
hdf.ncsa.uiuc.edu (141.142.21.14)
```

If you try to log on to NCSA's ftp server and receive the message saying too many users are connected to NCSA ftp, try this HDF ftp server.

Archive server: E-mail a request to: archive-server@ncsa.uiuc.edu Include in the subject or message line the word "help", then press RETURN. Send another e-mail request to: `archive-server@ncsa.uiuc.edu` Include in the subject or message line the word "index", then press RETURN. The information you receive from both the help and index commands will give you further instructions on obtaining NCSA software. Refer to Chapter 1 of *HDF Calling Interfaces Manual* for details.

US mail: A tape or CDROM archive of HDF is also available for purchase through the NCSA technical Resources Catalog. To obtain a catalog, contact:

```
NCSA Documentation Orders
152 Computing Applications Building
605 East Springfield Avenue Champaign, IL 61820
(217) 244-4130
```

B. Building the HDF Library Source.

The HDF base library can be built with a single command from the top level directory where the subdirectories "src/", "util/", and "test/" reside. The file "Makefile.template" is a generic, machine independent makefile which you can modify if there is no makefile already built for your machine.

For convenience, there are also machine-customized makefiles. For example. the "MAKE.IBM6000" file is a makefile suitable for compiling HDF on an IBM RS/6000. Assuming you are on an IBM RS/6000, copy "MAKE.IBM6000" to "Makefile" and use the following commands to install different targets:

```
cp MAKE.IBM6000 Makefile
```

`make allnofortran` builds the HDF library and only the C interfaces, the utilities and the C test programs. make all builds the HDF library with the C and Fortran-77 interfaces, the utilities, and C and Fortran-77 test programs.

General netCDF Installation Overview

Building the netCDF Library Source.

The HDF netCDF/HDF library build process is automatically configured by the makefile system. Modify the file named "CUSTOMIZE" in the "mfhdf" (which stands for ***multi-file hdf***) directory and run the script named "configure". It will set up all of the makefiles correctly. Refer to the "INSTALL", "README", and "README.HDF" files in the "mfhdf" directory for instruction and direction.

Procedures to Set Up the HDF Application Programming Environment

To use HDF routines in your C program, you must add the line `#include "hdf.h"` if you don't use the netCDF/HDF library, or `#include "mfhdf.h"` otherwise. This must be near the beginning of your code.

Note: Applications that need netCDF or multi-file SDS functionality should link with both "libnetcdf.a" and "libdf.a" **in this order** (the order is critical!). Applications that use neither of these interfaces can just link with the "libdf.a" library for the base level of HDF functionality.

If you are on a SUN SPARC, the include files are in the directory "incdir", the base library file "libdf.a" is in "libdir", and the netCDF/HDF library file "libnetcdf.a" is in "mflibdir". Use the following command to compile a C program "myprog.c":

```
cc -DSUN -DHDF -Iincdir myprog.c mflibdir/libnetcdf.a libdir/libdf.a -o myprog
```

or

```
cc -DSUN -DHDF -Iincdir myprog.c -L mflibdir -lnetcdf -L libdir -ldf -o myprog
```

The `mflibdir/libnetcdf.a` or `-L mflibdir -lnetcdf` need not be included if you are not using the multi-file interface.

For Fortran-77 programs, if your Fortran-77 compiler accepts 'include' statements, you may include :hdf.inc", "dffunc.inc", and "netcdf.inc" in your program. Otherwise, you need to declare in your program all the constants used and functions called by the program. To compile a Fortran-77 program "myprogf.f" use:

```
f77 -o myprogf myprogf.f mflibdir/libnetcdf.a libdir/libdf.a
```

or

```
f77 -o myprogf myprogf.f -L mflibdir -lnetcdf -L libdir -ldf
```

Again, the `mflibdir/libnetcdf.a` need not be included if you are not using the multifile interface.

Windows NT Installation

Please refer to the "install_winNT.txt" file in the "./release_notes" directory of the distribution.

Use of the Pablo Instrumentation of HDF

This version of the distribution has support for creating an instrumented version of the HDF library (libdf-inst.a). This library, along the Pablo performance data capture libraries, can be used to gather data about I/O behavior and procedure execution times.

More detailed documentation on how to use the instrumented version of the HDF library with Pablo can be found in the Pablo directory '$(toplevel)/hdf/pablo'. See the provided '$(toplevel)/hdf/pablo/README.Pablo' and the Postscript file '$(toplevel)/hdf/pablo/Pablo.ps'.

At this time only an instrumented version of the core HDF library libdf.a can be created. Future versions will have support for the SD interface found in libmfhdf.a. Current interfaces supported are AN, GR, DFSD, DFAN, DFP, DFR8, DF24, H, V, and VS.

To enable the creation of an instrumented library the following section in the makefile fragment($(toplevel)/config/mh-<os>) must be uncommented and set.

```
# ------------ Macros for Pablo Instrumentation  --------------------
# Uncomment the following lines to create a Pablo Instrumentation
# version of the HDF core library called 'libdf-inst.a'
# See the documentation in the directory 'hdf/pablo' for further
# information about Pablo and what platforms it is supported on
# before enabling.
# You need to set 'PABLO_INCLUDE' to the Pablo distribution
# include directory to get the files 'IOTrace.h' and 'IOTrace_SD.h'.
#PABLO_FLAGS  = -DHAVE_PABLO
#PABLO_INCLUDE = -I/hdf2/Pablo/Instrument.HP/include
```

After setting these values you must re-run the top-level 'configure' script. Make sure that your start from a clean re-build(i.e. 'make clean') after re-running the toplevel 'configure' script and then run 'make'. Details on running configure can be found in the section 'General Configuration/Installation - Unix' found in the top-level installation file '$(toplevel)/INSTALL'.

Joint Photographic Expert Group compression. See JPEG compression
RIS8. See 8-bit raster image data set
Run-length encoding compression. See RLE compression

Symbols

"Native format" option 20
"Simple" scientific data set
  description 26

Numerics

24-bit raster image 185–198
  compressing , data 192
  data representation 185
  determining the dimensions of a 194
  modifying the interlace mode of a 195
  querying the reference number of the most-recently-accessed 198
  querying the total number of , in a file 197
  reading 194–197
  reading a , with a given reference number 196
  routines for obtaining information about 197
  setting the interlace mode for a 190
  specifying that the next , read to be the first 197
  writing 188–??
24-bit Raster Image API
  description 3
24-bit raster image API 188
  routine categories 188
  routine list 188
24-bit raster image data set
  compression methods 186
  contents of a 185
  description 185
  interlace modes 187
  optional objects 186–188
  required objects 185–186
24-bit raster image data set dimension
  description 186
8-bit raster image
  compressing 176
  data representation 169, 170
  description 169
  determining the reference number of the most-recently-accessed 183
  determining the reference number of the palette of the most-recently-accessed 184
  querying the dimensions of a 181
  querying the total number of , in a file 183
  reading 180–183
  reading a , with a given reference number 182
  routines for obtaining information about 183
  specifying the next , to be read 183
  specifying the reference number for a 179
  writing 172–180
8-bit Raster Image API
  description 3
8-bit raster image API 172
  routine categories 172
  routine list 172
8-bit raster image data set 169–184
  compression methods 171
  description 169
  optional objects 170–172
  required objects 169–170
8-bit raster image data set data model 169–172
8-bit raster image data set dimension
  description 170
8-bit raster image data set palette
  description 170

A

AN API. See multifile annotation API
ANannlen
  description 271
  parameter list 272
ANannlist
  description 274
  parameter list 274
ANatype2tag
  description 277
  parameter list 277
ANcreate
  description 266
  parameter list 267
ANcreatef
  description 266
  parameter list 267
ANend

description 266
parameter list 267
ANendaccess
description 266
parameter list 267
ANfileinfo
description 271
parameter list 271
ANget_tagref
description 276
parameter list 276
ANid2tagref
description 276
parameter list 276
Annotation 247–276
creating an , using the multifile annotation API 266
creating and writing an , using the multifile annotation API 266–269
description 247
getting the length of an , using the multifile annotation API 271
obtaining a list of , corresponding to given search criteria using the multifile annotation API 274
obtaining information about an , using the multifile annotation API 271–??
obtaining information about every , in a file using the multifile annotation API 271
obtaining the number of , corresponding to given search criteria using the multifile annotation API 273
reading , using the single-file annotation API 255–260
reading an , using the multifile annotation API 270
selecting an , using the multifile annotation API 271
writing , using the single-file API 250–255
writing an , using the multifile annotation API 267
Annotation API
description 4
Annotation data model
description 247
ANnumann
description 273
parameter list 274
ANreadann
description 270
parameter list 270
ANselect
description 271
parameter list 271
ANstart
description 266
parameter list 267
ANtag2atype
description 277
parameter list 277
ANtagref2id
description 277
parameter list 277
ANwriteann
description 268
parameter list 268
Array rank 20
Attribute index
description 230

B

Buffer interlacing
description 113

C

Calibrated data
reading 94
writing 93
Calibration attribute 93–95
description 87, 93
Color lookup table
description 233
Command-line utilities 331–353
categories of 4
description 331
list of 331
purpose 331
COMP_JPEG define
description 186
COMP_NONE define
description 186
Composite image tags
list of 357
Compressing an HDF file 349

Compressing RIS8 images in an HDF file 347
Connectivity list
  description 143
Converting 24-bit raw raster images to RIS8 images 347
Converting 8-bit raster images to the HDF format 345
Converting an HDF RIS24 image to an HDF RIS8 image 348
Converting floating-point data to an SDS or RIS8 object 342
Converting raw palette data to the HDF palette format 348
Converting several RIS8 images to one 3D SDS 344
Coordinate system attribute
  description 87, 293

D

D24readref
  parameter list 197
Data element
  description 333
Data object
  description 333
Description annotation
  description 247
DF*lastref routine
  list and descriptions of all 261
  methods of determining a reference number through the use of a 261
DF24addimage
  description 189
  parameter list 189
DF24getdims
  description 194
  parameter list 195
DF24getimage
  description 194
  parameter list 195
DF24lastref
  description 198
  parameter list 198
DF24nimages
  description 197
  parameter list 198
DF24putimage
  description 189
  parameter list 189
DF24readref
  description 196
DF24reqil
  description 195
  parameter list 195
DF24restart
  description 197
  parameter list 197
DF24setcompress
  description 192
  parameter list 192
DF24setil
  description 190
  parameter list 192
DFAN API. See Single-file annotation API
DFANaddfds
  description 251
  parameter list 251
DFANaddfid
  description 251
  parameter list 251
DFANgetdesc
  description 258
  parameter list 259
DFANgetdesclen
  description 258
  parameter list 259
DFANgetfds
  description 256
  parameter list 256
DFANgetfdslen
  description 256
  parameter list 256
DFANgetfid
  description 255
  parameter list 256
DFANgetfidlen
  description 255
  parameter list 256
DFANgetlabel
  description 258
  parameter list 259

DFANgetlablen
  description 258
  parameter list 259
DFANlablist
  description 262
  parameter list 260, 263
DFANputdesc
  description 253
  parameter list 253
DFANputlabel
  description 252
  parameter list 253
DFPaddpal
  description 241
  parameter list 241
DFPgetpal
  description 243
  parameter list 243
DFPlastref
  description 245
DFPnpals
  description 244
  parameter list 245
DFPputpal
  description 241
  parameter list 241
DFPreadref
  description 244
  parameter list 244
DFPrestart
  description 244
DFPwriteref
  description 242
  parameter list 242
DFR8addimage
  description 173
  parameter list 173
DFR8getdims
  description 181
  parameter list 181
DFR8getimage
  description 180
  parameter list 181
DFR8getpalref
  description 184
DFR8lastref
  description 183
DFR8nimages
  description 183
  parameter list 183, 184
DFR8putimage
  description 173
  parameter list 173
DFR8readref
  description 182
  parameter list 183
DFR8restart
  description 183
DFR8setcompress
  description 176
  parameter list 176
DFR8setpalette
  description 174
  parameter list 175
DFR8writeref
  description 179
  parameter list 180
DFSD scientific data set 279–302
  assigning string attributes to a 293
  assigning value attributes to a 294
  contents of a 279
  creating a 282
  description 279
  determining the number of , in a file 288
  obtaining reference numbers for a 289
  optional objects 280
  preventing the reassignment of , attributes 285
  reading 286–289
  reading the attributes of a 297–299
  reading the dimension attributes of a 301–302
  reading the value attributes of a 298
  required objects 279
  resetting the default interlace settings for a 285
  specifying the data type of a 282
  specifying the dimensions and data type of a 287
  writing 281–286
  writing several 284
  writing the dimension attributes of a 299–301
  writing the dimension scale of a 300

DFSD scientific data set API
- description 3, 280
- routine categories 280
- routine list 281
- use of file identifiers in the 281
- use of predefined attributes in the 292–302
- use of slabs in the 290–292

DFSD scientific data set dimension
- writing the string attributes of a 299

DFSDadddata
- description 282
- parameter list 282

DFSDclear
- description 285
- parameter list 286

DFSDendslab
- description 290

DFSDgetcal
- description 299
- parameter list 299

DFSDgetdata
- description 286
- parameter list 286

DFSDgetdatalen
- description 297
- parameter list 297

DFSDgetdatastrs
- description 297
- parameter list 297

DFSDgetdimlen
- description 301
- parameter list 301

DFSDgetdims
- description 287
- parameter list 287

DFSDgetdimscale
- description 301
- parameter list 301

DFSDgetdimstrs
- description 301
- parameter list 301

DFSDgetfillvalue
- description 299
- parameter list 299

DFSDgetNT
- description 287
- parameter list 287

DFSDgetrange
- description 299
- parameter list 299

DFSDlastref
- description 289

DFSDndatasets
- description 288

DFSDputdata
- description 282
- parameter list 282

DFSDreadref
- description 289
- parameter list 289

DFSDreadslab
- description 292
- parameter list 292

DFSDrestart
- description 288

DFSDsetcal
- description 294
- parameter list 294

DFSDsetdatastrs
- description 293
- parameter list 293

DFSDsetdims
- description 285
- parameter list 285

DFSDsetdimscale
- description 300
- parameter list 301

DFSDsetdimstrs
- description 299
- parameter list 300

DFSDsetfillvalue
- description 294
- parameter list 294

DFSDsetlengths
- description 293, 299
- parameter list 293, 300

DFSDsetNT
- description 283
- parameter list 284

DFSDsetrange

description 294
parameter list 294
DFSDstartslab
description 290
parameter list 290
DFSDwriteref
description 284
parameter list 284
DFSDwriteslab
description 290
parameter list 291
Dimension attribute 81
Dimension compatbility mode
setting the current 73
Dimension compatibility mode
description 73
setting the future 73
Dimension format attribute
description 299
Dimension label attribute
description 299
Dimension scale
description 21
Dimension unit attribute
description 299
Displaying general information about the contents of an HDF file 350
Displaying vdata information 350

E

Editing the contents of an HDF file 333
Error reporting 303–??
Error reporting API
description 303
returning the code of the nth-most-recent error 304
returning the description of an error code 305
routine list 303
writing error stack information to a file 304
writing errors to a console window 305
External data file
creating a data set in a 42, 43
moving data to a 44
reading from a 52
specifying the directory search path in a 42
specifying the location of the next, to be created 43
writing to a 42
Extracting 8-bit raster images and palettes from HDF files 346
Extracting palette data from an HDF file 349

F

File annotation
comparison with object annotation 248
description 247
File attribute
description 81
File description annotation
assigning a , using the single-file annotation API 251
reading a , using the single-file annotation API 256
File interlacing
description 113
File label annotation
assigning a , using the single-file annotation API 251
reading a 270
reading a , using the single-file annotation API 255
fill mode
description 92
fill value
description 91
Fill value attribute 91–93
description 87
fill value attribute
reading a 92
writing a 92
FILL_ATTR define
description 227
Format attribute
description 87, 293
fptohdf 342

G

General raster image 199–238
accessing 203
compressing 225
creating 204
creating a , in an external file 226

getting the index of a 223
moving , to an external file 226
obtaining information about a 219
reading 210
reading from an external file 211
setting the I/O mode for external , access 227
setting the interlace mode for a 205
setting the interlace mode for a , or image read 211
terminating access to 203
writing 207
General raster image API
description 4
external file operations using the 226
mapping a reference number to an image index using the 238
mapping an image identifier to a reference number using the 238
object identifier conversion functions in the 238
obtaining a palette identifier using the 233
obtaining information about the contents of a file using , routines 218
obtaining palette information using the 233
reading and writing palette data using the 233–237
reading palette data using the 234
routine list 201
writing palette data using the 234
General raster image array
description 200
General raster image array name
description 200
General raster image attribute 227–233
predefined 227
querying user-defined 230
reading user-defined 230
setting user-defined 228
General raster image attributes
description 201
General raster image data model 199–201
General raster image data set
optional objects 200–201
required objects 199–200
General raster image data set API 201
programming model 202
routine categories 201
General raster image dimension
description 200
General raster image palettes
description 200
General raster image pixel type
data type 200
description 200
number-of-components 200
General raster image tags
list of 357
Global attribute
description 81
GRattrinfo
description 230
parameter list 231
GRcreate
description 205
parameter list 205
GRend
description 204
parameter list 204
GRendaccess
description 203
parameter list 204
GRfileinfo
description 218
parameter list 219
GRfindattr
description 230
parameter list 231
GRgetattr
description 230
parameter list 231
GRgetiminfo
description 219
parameter list 219
GRgetlutid
description 233
parameter list 235
GRgetlutinfo
description 233
parameter list 235
GRidtoref
description 238

parameter list 238
GRluttoref
description 234
parameter list 234
GRnametoindex
description 223
parameter list 223
Group object
description 333
GRreadimage
description 211
parameter list 211
GRreadlut
description 234
parameter list 235
GRreftoindex
description 223, 238
parameter list 223, 238
GRreqimageil
description 211
parameter list 211
GRreqlutil
parameter list 211
GRrequtil
description 211
GRselect
description 203
parameter list 204
GRsetaccesstype
description 227
parameter list 227
GRsetattr
description 228
parameter list 231
GRsetcompress
description 225
parameter list 225
GRsetexternalfile
description 226
parameter list 226
GRstart
description 203
parameter list 204
GRwriteimage
description 207
parameter list 208
GRwritelut
description 234
parameter list 235

H

HDF
description 1
purpose 1, 3
HDF API
description 2, 3
HDF installation
use of Pablo instrumentation 362
HDF installation overview 360–362
netCDF installation 361
setting up the application programming environment 361
Windows NT installation 362
HDF interface vs. netCDF interface 96–97
HDF_CHUNK_DEF union
definition of 59
hdf24hdf8 348
hdfcomp 347
hdfed 333
annotate command 339
close command 340
delete command 339
display command 340
dump command 338
getr8 command 341
help command 336
if conditional 342
info command 336
next command 338
open command 336
prev command 337
put command 341
putr8 command 340
revert command 341
select command 341, 342
wait command 342
write command 339
hdfed command set 335
hdfls 332

hdfpack 349
hdftopal 349
hdftor8 346
hdp 350
command set 350
dumprig command 352
dumpsds command 351
dumpvd command 351
dumpvg command 352
list command 351
HEprint
description 304
HEstring
description 305
HEvalue
description 304
history attribute
description 88
HXsetcreatedir
description 43
parameter list 43
HXsetdir
description 43
parameter list 43

I

Image compression. See IMCOMP compression
IMCOMP compression
description 172

J

JPEG compression
description 171, 187
JPEG compression quality factor
description 171

L

Label annotation
description 247
Label attribute
description 87, 293
Listing basic information about an HDF file 332
Local attribute
description 81
Lone vdata
description 126
Lone vgroup
description 156
Low-level interface 2
LUT. See Color lookup table

M

MFGR_INTERLACE_LINE define
description 205
MFGR_INTERLACE_PIXEL define
description 205
MFGR_INTERLACE_PLANE define
description 205
mgwcimg
definition of 208
mgwrimg
definition of 208
missing_value attribute
description 88
Multifile annotation API 264–276
description of the 264
list of tags used in 249
programming model for the 265
routine categories 264
routine list 265
type definitions specific to the 265

N

netCDF 95–97
netCDF data model
HDF support of 95
netCDF interface vs. HDF interface 96–97
Not-a-Number 92

O

Object annotation
comparison with file annotation 248
description 248
Object description annotation
assigning a , using the single-file annotation API 253
reading a , using the single-file annotation API 258
Object label annotation
assigning a , using the single-file annotation API 252
reading a , using the single-file annotation API

258
Obsolete tags
list of 358
Old and new dimension implementation 72

P

Palette 239–245
adding to a RIS8 object 174
backward compatibility issues 245
color mapping using a 239
description 239
obtaining the reference number of the most recently accessed 245
querying the number of , in a file 244
reading 242–244
reading a , with a given reference number 244
retrieving the reference number of the specified 234
specifying the next palette to be accessed to be the first 244
specifying the reference number of a 242
writing 240–242
Palette API
description 3, 240
routine categories 240
routine list 240
paltohdf 348
Performance Issues 311–??
Pixel interlacing
description 187
Pixels
description 169
Predefined attribute 87–95
accessing a 88
description 21, 87
list of , with labels and descriptions 87
list of parameters 88
naming conventions for 88
Predefined dimension string attribute
types of 299
Predefined string attribute
types of 293

R

r24hdf8 347
r8tohdf 345
Range attribute ??–91
description 87
reading a 91
writing a 91
Range attributes 91
Reference number
checking before assigning an object annotation 261
determining 261–264
determining the , for the last object accessed 261
querying a list of , for a given tag 262
Reference number, overwriting data for a specified 284
RGB values
description 185
RIS24. See 24-bit raster image data set
ristosds 344
RLE compression
description 171

S

Scan-line interlacing
description 187
Scan-plane interlacing
description 187
Scientific data set tags
list of 358
SD scientific data model
annotations and the 21
SD scientific data set 19–97
appending data to a 37
chunked 59
chunked , obtaining information about a 62
chunked , reading data from a 61
chunked , writing to a 61
chunks , setting the maximum number to cache 60
compressing data in a 40
contents 19, 71, 319, 320, 321, 322, 323
creating , with non-standard-length data 57
creating a 26
data type 20
default data representation 20
description 19

establishing access to a 24
locating a , by name 55
locating a , by reference number 55
making a chunked , from a generic 59
obtaining information about a 52
obtaining information about a specific 53
obtaining information about each , in a file 53
optional objects 20
reading from a 46
required objects 19
terminating access to a 25
writing to a 28
SD scientific data set API
compression methods supported 40
description 3, 21
programming model 23–25
routine categories 21–23
routine list 22
SD scientific data set array
description 20
SD scientific data set array name
description 20
SD scientific data set attribute 81
SD scientific data set dimension
description 20
naming a 72
obtaining information about a 74
selecting a 71
SD scientific data set dimension scales
writing 74
SD scientific data set dimension string attribute 90–91
reading a 90
writing a 90
SD scientific data set identifiers 23
SD scientific data set index
description 55
SD scientific data set string attribute
description 89
reading a 89
writing a 89
SD scientific data set tag 23
SDattrinfo
description 83
parameter list 84
SDcreate
description 26
parameter list 27
SDdiminfo
description 74
SDend
description 25
parameter list 25
SDendaccess
description 25
parameter list 25
SDexternalfile
description 44
SDfileinfo
description 53
parameter list 53
SDfindattr
description 83
parameter list 84
SDgetcal
description 94
parameter list 94
SDgetChunkInfo
description 62
parameter list 62
SDgetdatastrs
description 89
parameter list 89
SDgetdimid
description 71
SDgetdimstrs
description 90
parameter list 90
SDgetfillvalue
description 92
parameter list 93
SDgetinfo
description 53
parameter list 53
SDgetrange
description 91
parameter list 91
SDnametoindex
description 55

parameter list 56
SDreadattr
description 84
parameter list 84
SDreadChunk
description 61
parameter list 62
SDreaddata
description 46
parameter list 47
SDreftoindex
description 55
parameter list 56
SDS id
see SD scientific data set identifier 23
SDselect
parameter list 25
SDsetattr
description 82
parameter list 84
SDsetcal
description 93
parameter list 94
SDsetChunk
description 59
parameter list 59
SDsetChunkCache
description 60
parameter list 60
SDsetcompress
description 40
parameter list 41, 59, 60, 61, 62
SDsetdatastrs
description 89
SDsetdimname
description 72
parameter list 72
SDsetdimstrs
description 90
parameter list 90
SDsetdimval_bwcomp
description 73
parameter list 74
SDsetdimval_comp
description 73
parameter list 73
SDsetexternalfile
parameter list 44
SDsetfillmode
parameter list 93
SDsetfillvalue
description 92
parameter list 93
SDsetnbitdataset
description 57
parameter list 58
SDsetrange
description 91
parameter list 91
SDstart
description 24
SDwriteChunk
parameter list 61
SDwritechunk
description 61
SDwritedata
description 28
parameter list 30
writing data to chunked SDSs using 61
Self-description
definition 3
sfgcfill
description of 92
sfgfill
description of 92
sfrcatt
description of 84
sfrcdata
description of 46
sfrdata
description of 46
sfrnatt
description of 84
sfwcdata
description of 30
sfwdata
description of 30
Single-file Annotation API
routine categories 249
Single-file annotation API 248–264
list of tags used in 249

programming model for the 250
routine list 249
Slab
accessing a , using the single-file scientific data set API 290
description 29
reading a , using the single-file scientific data set API 291–292
writing , using the single-file scientific data set API 290–291
Strides
description 30
support of , in the single-file scientific data set API 291

T

title attribute
description 88

U

Unit attribute
description 87, 293
Unlimited dimension 20
User_defined attribute
allowed data types for a 81
writing a 81
User-contributed command-line utilities 352–353
list of 353
User-defined attribute 80–??
count 81
description 21, 80
index 81
naming rules 81
querying for a 83
reading a 84
Utility tags
list of 356

V

Vaddtagref
description 147
parameter list 148
Vattach
description 145
parameter list 146
Vattrinfo
description 159
parameter list 159
Vdata 101–??
attributes of a 129
creating 111
creating and writing to multifield 111–??
creating and writing to single-field 107–??
description 101
determining if the given , is an attribute 133
determining the reference number from a , name 127
obtaining information about a ??–135, 135–138, ??–163
querying information on a given , attribute 132
querying the number of attributes of a 131
querying the total number of , attributes 130
querying the values of a given , attribute 132
reading from a 122–126
resetting the current position within a 114
resetting the current record position within a 115
retrieving the index of a , attribute given the attribute name 131
searching for 126–129
searching for a , by field name 127
searching for lone , 126
searching for the reference number of a 127
selecting the set of , to be read 123
setting the attribute of a , 130
storing data in multiple , of mixed data types 118
writing buffers into 116
writing to a 115
writing to a multifield 114
Vdata API
description 4
Vdata class
assigning to a vdata 112
Vdata data model
description 101
Vdata field
defining 112
description of a predefined 112
initializing for write access 113

querying the index of a , given the field name 129
querying the number of attributes of a 131
removing alignment bytes when writing to a 119
setting the attribute of a , 130
Vdata identifier
determining the next 166
Vdata interlace mode
specifying the 113
Vdata name
assigning to a vdata 112
Vdetach
description 146
parameter list 146
Vend
description 146
parameter list 146
Vertex set. See vset
VF field information retrieval routine set 140
VFfieldesize
description 140
parameter list 140
VFfieldisize
description 140
parameter list 140
VFfieldname
description 140
parameter list 140
VFfieldorder
description 140
parameter list 140
VFfieldtype
description 140
parameter list 140
Vfindattr
description 161
parameter list 161
VFnfields
description 140
parameter list 140
Vgetattr
description 160
parameter list 160
Vgetclass
description 156
parameter list 156, 164
Vgetid
description 156
parameter list 156
Vgetname
description 156
parameter list 156, 164
Vgetnext
description 166
parameter list 167
Vgettagref
description 163
Vgettagrefs
description 163
Vgetversion
description 159
parameter list 159
Vgroup 141–167
accessing a 145
assigning a class to a 147
assigning a name to a 147
attributes of a 158
command-line utilities for 166
containing two RIS8 objects and a vdata 142
conventions on content and structure 142
creating a 146
creating and writing to a 146–154
description 141
determining the name of a 167
inserting a HDF object into a 147
inserting a vdata or vgroup into a 148
locating a 155
locating a lone 156
means of uniquely identifying a 144
obtaining information about the contents of a 163–166
organization of 142
querying information on a specified , attribute 159
querying the number of , members 167
querying the total number of , attributes 160
querying the values of a given , attribute 160
querying the version of a given , 159
reading from a 155–158
retrieving the index of a , attribute given the attribute name 160

returning , member information 163
returning the tag/reference number pairs of , contents 163
setting the attribute of a 160
sharing of vgroups and vdatas between more than one 142
similarity to the Unix file system 141
terminating access to 146
Vgroup API 143–144
description 4
obsolete routines 166–167
routine categories 143
routine list 144
Vgroup API programming model 145–146
Vgroup class
description 142
determining a 156
Vgroup identifier 144
determining the next 156, 166
Vgroup name
description 141
determining a 156
Vinqtagref
description 163
Vinquire
description 167
parameter list 167
Vinsert
description 148
parameter list 148
Visvg
description 163
parameter list 164
Visvs
description 163
parameter list 164
Vlone
description 156
parameter list 156
Vmake
description 166
Vnattrs
description 160
parameter list 160
Vntagrefs
description 163
VS vdata information retrieval routine set 139
VSattach
creating vdatas with 112
VSattrinfo
description 132
parameter list 132
VSelts
description 139
parameter list 139
Vset
describing a heated mesh 143
description 142
structure 143
Vset node
description 143
Vset tags
list of 358
Vsetattr
description 160
parameter list 160
Vsetclass
description 147
parameter list 148
Vsetname
description 147
parameter list 148
VSfdefine
description 112
parameter list 113
VSfexist
description 127
parameter list 127
VSfind
description 127
parameter list 127
VSfindattr
description 131
parameter list 131
VSfindex
description 129
parameter list 130
VSfnattrs
description 131
parameter list 131

VSgetattr
  description 132
  parameter list 132
VSgetclass
  description 139
  parameter list 139
VSgetfields
  description 139
  parameter list 139
VSgetid
  description 127
  parameter list 127
VSgetinterlace
  description 139
  parameter list 139
VSgetname
  description 139
  parameter list 139
Vshow
  description 166
vshow 350
VSinquire
  description 136
  parameter list 136
VSisattr
  description 133
  parameter list 133
VSlone
  description 126
  parameter list 127
VSnattrs
  description 130
  parameter list 130
VSQuery vdata information retrieval routine set 138
VSQuerycount
  parameter list 139
VSQueryfields
  description 138
  parameter list 138
VSQueryinterlace
  description 138
  parameter list 138
VSQueryname
  description 138
  parameter list 138
VSQueryref
  description 138
  parameter list 139
VSQuerytag
  description 138
  parameter list 139
VSQueryvsize
  description 138
  parameter list 139
VSread
  description 123
  parameter list 124
  setting the file interlace mode using 113
VSseek
  description 123
  misused to append data 115
  parameter list 114, 116, 124
VSsetattr
  description 130
  parameter list 130
VSsetclass
  description 112
  parameter list 113
VSsetfields
  description 113, 123
  parameter list 113, 124
VSsetinterlace
  description 113
  parameter list 113
VSsetname
  description 112
  parameter list 113
VSsizeof
  description 139
  parameter list 139
Vstart
  description 145
  parameter list 146
VSwrite
  description 115
  parameter list 116
  setting the file interlace mode using 113