Starlink Project
Starlink User Note 33.12

R.F. Warren-Smith & D.S. Berry

30st April 2020

NDF

Routines for Accessing the Extensible N-Dimensional Data Format Version 2.1 Programmer's Manual

---

Abstract

The *Extensible N-Dimensional Data Format* (NDF) is a format for storing bulk data in the form of N-dimensional arrays of numbers. It is typically used for storing spectra, images and similar datasets with higher dimensionality. The NDF format is based on the *Hierarchical Data System* HDS (SUN/92) and is extensible; not only does it provide a comprehensive set of standard ancillary items to describe the data, it can also be extended indefinitely to handle additional user-defined information of any type.

This document describes the routines provided for accessing NDF data objects. It also discusses all the important NDF concepts and includes a selection of simple example applications. The majority of the text describes the Fortran 77 interface for the NDF library. The C interface is described briefly in an appendix.

Contents

1 INTRODUCTION 1
1.1 What is an NDF? 1
1.2 The NDF Philosophy and Extensibility 1
1.3 The Relationship with HDS 2
1.4 Background Reading 2
1.5 Scope of the Current Implementation 2

2 OVERVIEW 3
2.1 Overview of an NDF 3
2.2 Overview of the NDF_ Routines 5
2.3 Error Handling 6
2.4 Overview of a Typical Application 6

3 OBTAINING AND USING NDF IDENTIFIERS 8
3.1 Accessing NDFs for Input 8
3.2 NDF Identifiers 10
3.3 Annulling Identifiers 10
3.4 NDF Contexts: BEGIN and END 11
3.5 Cloning Identifiers 11

4 NDF SHAPE AND SIZE INFORMATION 11
4.1 Dimensionality and Bounds 12
4.2 Dimension Sizes 13
4.3 NDF Size 14
4.4 "Safe" Dimension Sizes under Error Conditions 14

5 GENERAL PROPERTIES OF NDF COMPONENTS 15
5.1 Specifying Component Names 15
5.2 Component State 15
5.3 Factors Determining a Component's State 16
5.4 Restrictions on the Data Component's State 16

6 ACCESSING CHARACTER COMPONENTS 17
6.1 Reading and Displaying Character Values 17
6.2 Writing Character Values 17
6.3 Character Component Length 18
6.4 Resetting Character Components 18

7 ARRAY COMPONENT TYPES 18
7.1 Numeric Types 19
7.2 Complex Values 20
7.3 Full Type Specifications 20
7.4 Setting Component Types 21

8 ACCESSING ARRAY COMPONENTS 22
8.1 Overview of Mapped Access to Array Components 22
8.2 Mapping and Unmapping 22

8.3 Implicit Unmapping . . . 24
8.4 Writing and Modifying Array Component Values . . . 25
8.5 More About Mapping Modes . . . 25
8.6 Initialisation Options . . . 26
8.7 Implicit Type Conversion . . . 26
8.8 Accessing Complex Values . . . 27
8.9 Mapping the Variance Component as Standard Deviations . . . 28
8.10 Restrictions on Mapped Access . . . 29

9 BAD PIXELS 29
9.1 The Need for Bad Pixels . . . 29
9.2 Recognition and Processing of Bad Pixels . . . 30
9.3 The Bad-Pixel Flag . . . 31
9.4 Obtaining and Using the Bad-Pixel Flag . . . 32
9.5 Requesting Explicit Checks for Bad Pixels . . . 33
9.6 Setting the Bad-Pixel Flag . . . 34
9.7 Interaction with Mapping and Unmapping . . . 35
9.8 Interaction with Initialisation Options . . . 35
9.9 A Practical Template for Handling the Bad-Pixel Flag . . . 36

10 THE QUALITY COMPONENT IN MORE DETAIL 37
10.1 Purpose of the Quality Component . . . 37
10.2 Accessing the Quality Array Directly . . . 37
10.3 The Bad-bits Mask . . . 38
10.4 Why Ignoring the Quality Component Works . . . 40
10.5 Controlling Automatic Quality Masking . . . 41

11 EXTENSIONS 42
11.1 Extensibility . . . 42
11.2 Extension Names and Software Packages . . . 43
11.3 The Contents of Extensions . . . 43
11.4 Accessing Existing Extensions . . . 43
11.5 Creating New Extensions . . . 45
11.6 Accessing Array Information in Extensions . . . 46
11.7 Deleting Extensions . . . 47
11.8 Enumerating an NDF's Extensions . . . 47

12 ARRAY COMPONENT STORAGE FORM AND COMPRESSION 48
12.1 General . . . 48
12.2 Obtaining the Storage Form . . . 48
12.3 Simple Storage Form . . . 48
12.4 Scaled Storage Form . . . 49
12.5 Delta compressed Storage Form . . . 50
12.6 Primitive Storage Form . . . 51

13 ACCESSING NDFS FOR OUTPUT 52
13.1 Using Existing NDFs . . . 52
13.2 Creating New NDFs via Parameters . . . 52
13.3 Conditional NDF Creation . . . 53

13.4 Creating Primitive NDFs 53

14 COMPONENT PROPAGATION 53
14.1 General 53
14.2 Propagation Rules for Standard NDF Components 54
14.3 Propagation Rules for Extensions 55
14.4 Creating New NDFs by Propagation 56
14.5 Default Propagation of Components and Extensions 56
14.6 Forcing Component Propagation 56
14.7 Inhibiting Component Propagation 57
14.8 Controlling Propagation of Extensions 57

15 NDF SECTIONS 57
15.1 The Need for NDF Sections 57
15.2 Creating NDF Sections 58
15.3 The Distinction between Base NDFs and Sections 58
15.4 Referring to Subsets and Super-sets 59
15.5 The Transfer Window 59
15.6 Changing Dimensionality 60
15.7 Restrictions on the Use of Sections 61
15.8 Restrictions on Mapped Access to Sections 61
15.9 More Advanced Use: Partitioning 62
15.10 Chunking 62
15.11 Blocking 64

16 USING SUBSCRIPTS TO ACCESS NDF SECTIONS 65
16.1 Specifying Lower and Upper Bounds 66
16.2 Specifying Centre and Extent 67
16.3 Using WCS Coordinates to Specify Sections 67
16.4 Using Normalised Pixel Coordinates to Specify Sections 68
16.5 Changing Dimensionality 69
16.6 Mixing Bounds Expressions 69

17 MERGING AND MATCHING NDF ATTRIBUTES 70
17.1 The Problem 70
17.2 Merging and Matching Bad-Pixel Flags 70
17.3 Matching NDF Bounds 73
17.4 Merging and Matching Numeric Types 75

18 THE AXIS COORDINATE SYSTEM 77
18.1 Pixel Coordinates 78
18.2 Axis Coordinates 78
18.3 Axis Arrays 79
18.4 Pixel Positions and Dimensions 79
18.5 Default Axis Array Values 80
18.6 Contiguous and Non-Contiguous Pixels 80
18.7 Processing Axis Array Values 81
18.8 Axis Normalisation 82

19 AXIS COMPONENTS 82
19.1 Overview of an NDF's Axis Components . . . 82
19.2 Axis Component States . . . 84
19.3 Restrictions on Axis Component States . . . 84
19.4 Defining a Default Axis Coordinate System . . . 85
19.5 Resetting Axis Components . . . 85
19.6 Accessing Axis Character Components . . . 86
19.7 Mapping Axis Arrays for Reading . . . 87
19.8 Unmapping Axis Arrays . . . 88
19.9 Writing and Modifying Axis Arrays . . . 89
19.10 Accessing Axis Variance Values as Standard Deviations . . . 90
19.11 Axis Normalisation Flags . . . 90
19.12 The Numeric Type of Axis Arrays . . . 91
19.13 The Storage Form of Axis Arrays . . . 91
19.14 Accessing Axis Components via NDF Sections . . . 92
19.15 Axis Extrapolation . . . 92

20 CONNECTING WITH THE DATA SYSTEM 93
20.1 NDF Names . . . 93
20.2 Finding and Importing NDFs . . . 94
20.3 A Note on Modes of Access . . . 95
20.4 Obtaining an HDS Locator for an NDF . . . 95
20.5 NDF Placeholders . . . 96
20.6 Creating NDFs via Placeholders . . . 97
20.7 Temporary NDFs . . . 97
20.8 Copying NDFs . . . 98
20.9 Selective Copying of NDF Components . . . 98
20.10 General Access to NDFs . . . 98
20.11 Deleting NDFs . . . 99

21 ALTERING BOUNDS AND PIXEL INDICES 99
21.1 Setting New Pixel-Index Bounds and Dimensionality . . . 99
21.2 Applying Pixel-Index Shifts . . . 100

22 THE HISTORY COMPONENT 101
22.1 Purpose of the History Component . . . 101
22.2 The History Component's State . . . 102
22.3 Preparing to Record History Information . . . 102
22.4 Resetting the History Component . . . 102
22.5 Default History Recording . . . 103
22.6 Propagation of History Information . . . 104
22.7 Explicitly Controlling History Text . . . 104
22.8 The History Update Mode . . . 106
22.9 Using Message Tokens in History Text . . . 107
22.10 History Text Width . . . 107
22.11 Formatting Options for History Text . . . 108
22.12 Automatic Error Recording . . . 108
22.13 Enquiring about Past History Information . . . 109

22.14 Accessing History by Date and Time . . . 109
22.15 Accessing Past History Text . . . 110
22.16 Modifying Past History . . . 111
22.17 Naming an Application . . . 112
22.18 Ending an Application . . . 113

23 MISCELLANEOUS FACILITIES 114
23.1 Restricting Access via NDF Identifiers . . . 114
23.2 Message System Routines . . . 115
23.3 Tuning the NDF_ System . . . 115

24 COMPILING AND LINKING 117
24.1 Standalone Applications . . . 117
24.2 ADAM Applications . . . 118

A EXAMPLE APPLICATIONS 119
A.1 SHOW — Display the size of an NDF . . . 119
A.2 SETTITLE — Assign a New NDF Title . . . 121
A.3 GETMAX — Obtain the Maximum Pixel Value . . . 123
A.4 GETSUM — Sum the Pixel Values . . . 126
A.5 READIMG — Read an image into an NDF . . . 130
A.6 ZAPPIX — "Zap" Prominent Pixels in an Image . . . 135
A.7 ADD — Add Two NDF Data Structures . . . 140
A.8 NDFTRACE — Trace an NDF Structure . . . 144

B ALPHABETICAL LIST OF FORTRAN ROUTINES 152

C CLASSIFIED LIST OF FORTRAN ROUTINES 159
C.1 Access to Existing NDFs . . . 159
C.2 Enquiring NDF Attributes . . . 159
C.3 Enquiring Component Attributes . . . 160
C.4 Creating and Deleting NDFs . . . 160
C.5 Setting NDF Attributes . . . 161
C.6 Setting Component Attributes . . . 161
C.7 Access to Component Values . . . 161
C.8 Enquiring and Setting Axis Attributes . . . 162
C.9 Access to Axis Values . . . 162
C.10 Access to World Coordinate System Information . . . 163
C.11 Creation and Control of Identifiers . . . 163
C.12 Handling NDF (and Array) Sections . . . 163
C.13 Matching and Merging Attributes . . . 164
C.14 Parameter System Routines . . . 164
C.15 Message System Routines . . . 165
C.16 Creating Placeholders . . . 165
C.17 Copying NDFs . . . 165
C.18 Handling Extensions . . . 165
C.19 Handling History Information . . . 166
C.20 Tuning the NDF_ system . . . 167
C.21 Compression . . . 167

D FORTRAN ROUTINE DESCRIPTIONS 168
NDF_ACGET . . . 169
NDF_ACLEN . . . 170
NDF_ACMSG . . . 171
NDF_ACPUT . . . 172
NDF_ACRE . . . 173
NDF_AFORM . . . 174
NDF_AMAP . . . 175
NDF_ANNUL . . . 176
NDF_ANORM . . . 177
NDF_AREST . . . 178
NDF_ASNRM . . . 179
NDF_ASSOC . . . 180
NDF_ASTAT . . . 181
NDF_ASTYP . . . 183
NDF_ATYPE . . . 184
NDF_AUNMP . . . 185
NDF_BAD . . . 186
NDF_BASE . . . 187
NDF_BB . . . 188
NDF_BEGIN . . . 189
NDF_BLOCK . . . 190
NDF_BOUND . . . 191
NDF_CANCL . . . 192
NDF_CGET . . . 193
NDF_CHUNK . . . 194
NDF_CINP . . . 195
NDF_CLEN . . . 196
NDF_CLONE . . . 197
NDF_CMPLX . . . 198
NDF_CMSG . . . 200
NDF_COPY . . . 201
NDF_CPUT . . . 203
NDF_CREAT . . . 204
NDF_CREP . . . 205
NDF_CREPL . . . 206
NDF_DELET . . . 207
NDF_DIM . . . 208
NDF_END . . . 209
NDF_EXIST . . . 210
NDF_FIND . . . 211
NDF_FORM . . . 212
NDF_FTYPE . . . 213
NDF_GTDLT . . . 214
NDF_GTSZx . . . 215
NDF_GTUNE . . . 216
NDF_GTWCS . . . 217
NDF_HAPPN . . . 218

NDF_HCOPY . . . 219
NDF_HCRE . . . 220
NDF_HDEF . . . 221
NDF_HECHO . . . 222
NDF_HEND . . . 223
NDF_HFIND . . . 224
NDF_HGMOD . . . 225
NDF_HINFO . . . 226
NDF_HNREC . . . 228
NDF_HOUT . . . 229
NDF_HPURG . . . 230
NDF_HPUT . . . 231
NDF_HSDAT . . . 233
NDF_HSMOD . . . 234
NDF_ISACC . . . 235
NDF_ISBAS . . . 236
NDF_ISIN . . . 237
NDF_ISTMP . . . 238
NDF_LOC . . . 239
NDF_MAP . . . 240
NDF_MAPQL . . . 242
NDF_MAPZ . . . 243
NDF_MBAD . . . 244
NDF_MBADN . . . 245
NDF_MBND . . . 246
NDF_MBNDN . . . 247
NDF_MSG . . . 248
NDF_MTYPE . . . 249
NDF_MTYPN . . . 251
NDF_NBLOC . . . 252
NDF_NCHNK . . . 253
NDF_NEW . . . 254
NDF_NEWP . . . 255
NDF_NOACC . . . 256
NDF_OPEN . . . 257
NDF_PLACE . . . 259
NDF_PROP . . . 260
NDF_PTSZx . . . 262
NDF_PTWCS . . . 263
NDF_QMASK . . . 264
NDF_QMF . . . 266
NDF_RESET . . . 267
NDF_SAME . . . 268
NDF_SBAD . . . 269
NDF_SBB . . . 270
NDF_SBND . . . 271
NDF_SCOPY . . . 272
NDF_SCTYP . . . 274

NDF_SECT . . . 275
NDF_SHIFT . . . 276
NDF_SIZE . . . 277
NDF_SQMF . . . 278
NDF_SSARY . . . 279
NDF_STATE . . . 280
NDF_STYPE . . . 281
NDF_TEMP . . . 282
NDF_TUNE . . . 283
NDF_TYPE . . . 285
NDF_UNMAP . . . 286
NDF_VALID . . . 287
NDF_XDEL . . . 288
NDF_XIARY . . . 289
NDF_XGT0x . . . 291
NDF_XLOC . . . 293
NDF_XNAME . . . 295
NDF_XNEW . . . 296
NDF_XNUMB . . . 298
NDF_XPT0x . . . 299
NDF_XSTAT . . . 300
NDF_ZDELT . . . 301
NDF_ZSCAL . . . 303

E THE NDF_ LIBRARY C INTERFACE 305
E.1 Conventions Used in the C Interface . . . 305
E.2 Multi-threaded Applications . . . 306
E.3 C-only Functions . . . 307
ndfLock . . . 308
ndfLocked . . . 309
ndfReport . . . 310
ndfUnlock . . . 311
E.4 Building C Applications . . . 312

F ALPHABETICAL LIST OF C FUNCTIONS 313

G OBSOLETE ROUTINES 336
NDF_IMPRT . . . 337
NDF_TRACE . . . 338

H CHANGES AND NEW FEATURES 339
H.1 Changes Introduced in V1.3 . . . 339
H.2 Changes Introduced in V1.4 . . . 340
H.3 Changes Introduced in V1.5 . . . 341
H.4 Changes Introduced in V1.6 . . . 341
H.5 Changes Introduced in V1.7 . . . 341
H.6 Changes Introduced in V1.8 . . . 341
H.7 Changes Introduced in V1.9 . . . 342
H.8 Changes Introduced in V1.10 . . . 342

H.9 Changes Introduced in V1.11 342
H.10 Changes Introduced in V1.12 343
H.11 Changes Introduced in V1.13 343
H.12 Changes Introduced in V2.0 343
H.13 Changes Introduced in V2.1 343
H.14 Changes Introduced in V2.2 344

1 INTRODUCTION

1.1 What is an NDF?

"NDF" stands for *Extensible N-Dimensional Data Format*. It is a standard file format for storing data which represent N-dimensional arrays of numbers, such as spectra, images, *etc*. and it therefore forms the basis of many spectral and image-processing applications. This document describes a subroutine library for accessing data stored in this form from applications written to run within a programming environment (such as ADAM – see SUN/101 & SG/4). For applications which do not use such facilities, a "stand-alone", version of the library is also available (see §24).

1.2 The NDF Philosophy and Extensibility

The main reason for using NDF data structures as a standard method of storing astronomical data is to simplify the exchange of information between separate applications packages. In principle, this should make it possible for a software user to process the same data using software drawn from any package.

In practice, previous attempts to define a standard data format for this purpose have met with two serious obstacles. First, different authors of software have interpreted the meaning of data items differently, so that although several software packages might be capable of reading the same data files, the different packages actually performed incompatible operations on the data. Secondly, many software authors have found a pre-defined data format to be too restrictive, and have simply chosen not to use it.

The NDF data structure has therefore had to satisfy two apparently quite contradictory requirements:

(1) Its interpretation should be closely defined, so that different (usually geographically separated) programmers can write software which processes it in consistent and mutually compatible ways.

(2) It should be very adaptable, so that it can be used to hold data associated with a wide variety of software systems, whose detailed requirements may vary considerably.

The solution to this problem has been to introduce the concept of *extensibility*, and to divide the NDF data structure into two parts – a set of *standard components* and a set of *extensions* – each of which individually satisfies one of these two requirements. An NDF data structure therefore consists of a central "core" of information, whose interpretation is well-defined and for which general-purpose software can be written with wide applicability, together with an arbitrary number of extensions which may be used by specialist software but whose interpretation is not otherwise defined. Those who wish to know more of the background to this philosophy can find a detailed discussion in SGP/38.

Most of the present document is concerned with the facilities provided by the fixed "core" of standard components, although a few general routines for handling extensions are also included.

1.3 The Relationship with HDS

The NDF file format is based upon the Hierarchical Data System HDS (SUN/92) and NDF data structures are stored in HDS *container files* (which by convention have a file type of .sdf). However, this does not necessarily mean that all applications which can read HDS files can also handle data stored in NDF format.

To understand why, you must appreciate that HDS provides only a rather low-level set of facilities for storing and handling astronomical data. These include the ability to store primitive data objects (such as arrays of numbers, character strings, *etc.*) in a convenient and self-describing way within *container files*. However, the most important aspect of HDS is its ability to group these primitive objects together to build larger, more complex structures. In this respect, HDS can be regarded as a construction kit which other higher-level software can use to build even more sophisticated data formats.

The NDF is a higher-level data format which has been built in this way out of the more primitive facilities provided by HDS. Thus, in HDS terms, an NDF is a data structure constructed according to a particular set of conventions to facilitate the storage of typical astronomical data (such as spectra, images, or similar objects of higher dimensionality).

While HDS can be used to access such structures, it does not contain any of the interpretive knowledge needed to assign astronomical meanings to the various components of an NDF, whose details can become quite complicated. In practice it is therefore cumbersome to process NDF data structures using HDS directly. Instead, the NDF access routines described in this document are provided. These routines "know" about how NDF data structures are build within HDS, so they can hide these details from the writers of astronomical applications. This results in a subroutine library which deals in higher-level concepts more closely related to the work which typical astronomical applications need to perform, and which emphasises the data concepts which an NDF is designed to represent, rather than the details of its implementation.

1.4 Background Reading

It is assumed that the reader of this document has some background knowledge of the programming environment being used (*e.g.* see SUN/101 & SG/4 for details of ADAM) and with the basic concepts used by the HDS data system (SUN/92).

A knowledge of the method by which the NDF is implemented using HDS (SGP/38) may also be useful as background reading, but is not essential for understanding the present document which is intended to be self-contained in its description of NDF concepts and facilities.

1.5 Scope of the Current Implementation

A small number of the basic facilities described in SGP/38 still remain to be supported, although most of the major facilities are now available through the interface described here. In addition, the WCS component has been introduced, going beyond the original NDF design in SGP/38. In cases of doubt, the presence or absence of a description of an NDF facility in this document should be taken as indicating the extent of the current implementation.

2 OVERVIEW

This section presents an overview of what NDF data structures are, and the facilities which the NDF_ system provides for manipulating them.

2.1 Overview of an NDF

The simplest way of regarding an NDF is to view it as a collection of those items which might typically be required in an astronomical image or spectrum. The main part is an N-dimensional array of *data* (where N is 1 for a spectrum, 2 for an image, *etc.*), but this may also be accompanied by a number of other items which are conveniently categorised as follows:

Character components:	**TITLE**	— NDF title
	LABEL	— Data label
	UNITS	— Data units
Array components:	**DATA**	— Data pixel values
	VARIANCE	— Pixel variance estimates
	QUALITY	— Pixel quality values
Miscellaneous components:	**AXIS**	— Coordinate axes
	WCS	— World coordinate systems
	HISTORY	— Processing history
Extensions:	**EXTENSION**	— Provides extensibility

The names of these components are significant, since they are used by the NDF access routines to identify the component(s) to which certain operations should be applied.[1] The following describes the purpose and interpretation of each component in slightly more detail.

Character components:

TITLE – This is a character string, whose value is intended for general use as a heading for such things as graphical output; *e.g.* 'M51 in good seeing'.

LABEL – This is a character string, whose value is intended to be used on the axis of graphs to describe the quantity in the NDF's *data* component; *e.g.* 'Surface brightness'.

UNITS – This is a character string, whose value describes the physical units of the quantity stored in the NDF's *data* component; *e.g.* 'J/(m∗∗2∗Ang∗s)'.

Array components:

[1] Note that the name "DATA" used by the NDF_ routines to refer to an NDF's *data* component differs from the actual name of the HDS object in which it is stored, which is "DATA_ARRAY".

DATA – This is an N-dimensional array of pixel values representing the spectrum, image, *etc.* stored in the NDF. This is the only NDF component which must always be present. All the others are optional.

VARIANCE – This is an array of the same shape and size as the *data* array, and represents the measurement errors or uncertainties associated with the individual *data* values. If present, these are always stored as *variance* estimates for each pixel.

QUALITY – This is an array of the same shape and size as the *data* array, and holds a set of unsigned byte values. These are used to assign additional "quality" attributes to each pixel (for instance, whether it is part of the active area of a detector). Quality values may be used to influence the way in which the NDF's *data* and *variance* components are processed, both by general-purpose software and by specialised applications.

Miscellaneous components:

AXIS – This component name represents a group of *axis* components which may be used to describe the shape and position of the NDF's pixels in a rectangular coordinate system. The physical units and a label for each axis of this coordinate system may also be stored. (Note that the ability to associate *extensions* with an NDF's *axis* coordinate system, although described in SGP/38, is not yet available via the NDF access routines described here.)

WCS – This component may be used to hold information about any "world coordinate systems" associated with the NDF. These may include celestial coordinate systems, such as right ascension and declination (in various flavours), but may also represent other coordinates, including wavelength.[2] Multiple coordinate systems may be present.

The WCS component is a rather more complex entity than most other NDF components and a full description is currently beyond the scope of this document. It stores world coordinate information in a format defined by the AST library (see SUN/210) and known as a "FrameSet". You should consult SUN/210 for a full description of the facilities which a FrameSet provides. The NDF_ library simply provides routines for reading and writing this information (see NDF_GTWCS and NDF_PTWCS).

HISTORY – This component may be used to keep a record of the processing history which the NDF undergoes. If present, this component should be updated by any applications which modify the data structure.

[2] In this respect, the WCS component provides a superset of the facilities provided by the AXIS component. However, the AXIS component is retained because it has been used historically by a significant number of astronomical applications. The NDF_ library maintains consistency between these two components (to the extent that their nature allows). Thus, for example, the rectangular coordinate system defined by the AXIS component is also accessible through the WCS component.

Extensions:

EXTENSIONs are user-defined HDS structures associated with the NDF, and are used to give the data format flexibility by allowing it to be extended. Their formulation is not covered by the NDF definition, but a few simple routines are provided for accessing and manipulating named extensions, and for reading and writing the values of components stored within them.

2.2 Overview of the NDF_ Routines

The NDF access routines described in this document all have names of the form:

NDF_<name>

where <name> identifies the operation which the routine performs. These routines provide facilities for performing the following types of operation on NDF data structures:

- Obtaining access to NDFs, for both input and output.
- Creating and deleting NDFs.
- Enquiring about the attributes of NDFs, including their shape and size.
- Enquiring about the attributes of NDF components.
- Reading, writing and resetting NDF component values.
- Enquiring about (and flagging) the presence of *bad* pixels in NDF components.
- Accessing and handling *quality* information associated with NDFs.
- Modifying the attributes of NDFs (including their shape and size) and the attributes of their components (such as their numeric type).
- Reading, writing and resetting the values of *axis* arrays and other *axis* components associated with NDFs.
- Modifying the attributes of NDF *axis* components (such as the numeric type of *axis* arrays).
- Controlling the propagation of NDF components to output data structures.
- Creating, deleting and enquiring about NDF extensions, and obtaining access to components stored within extensions.
- Controlling the propagation of NDF extensions to output data structures.
- Selection and management of *sections* which refer to subsets or super-sets of NDFs.
- Merging the attributes of NDFs to match the processing capabilities of specific applications.
- Importing and finding NDFs held in HDS container files and copying of NDFs between different HDS locations.

- Recording, accessing and deleting information about the processing history of an NDF.
- Constructing messages about NDFs.
- Controlling access to NDFs.

A full description of each routine can be found in Appendix D of this document.

2.3 Error Handling

The NDF_ routines adhere throughout to the standard error-handling strategy described in SUN/104. Most of the routines therefore carry an integer inherited status argument called STATUS and will return without action unless this is set to the value SAI__OK[3] when they are invoked. When necessary, error reports are made through the ERR_ routines in the manner described in SUN/104. Where exceptions to this general behaviour exist, they are noted in the appropriate subroutine descriptions in Appendix D.

2.4 Overview of a Typical Application

The following contains an example of a simple application which uses the NDF_ routines to add the *data* arrays of two NDF data structures to produce a new NDF. This is not quite the simplest "add" application which could be written, but is close to it. Nevertheless, it will do a good job, and will respond correctly to unforseen circumstances or conditions which it is not designed to handle by issuing sensible error messages.

The intention here is simply to give a flavour of how the NDF_ routines are used, so don't worry if you don't understand all the details. The example is followed by some brief programming notes which include references to other relevant sections of this document which can be consulted if necessary. If you are interested, a more sophisticated "add" application with extra commentary can also be found in §A.7.

```
      SUBROUTINE ADD( STATUS )                                [1]
      INCLUDE 'SAE_PAR'                                       [2]
      INTEGER STATUS, EL, NDF1, NDF2, NDF3, PNTR1( 1 ), PNTR2( 1 ), PNTR3( 1 )

*  Check inherited global status and begin an NDF context.
      IF ( STATUS .NE. SAI__OK ) RETURN                       [3]
      CALL NDF_BEGIN                                          [4]

*  Obtain identifiers for the two input NDFs and trim their pixel-index
*  bounds to match.
      CALL NDF_ASSOC( 'IN1', 'READ', NDF1, STATUS )           [5]
      CALL NDF_ASSOC( 'IN2', 'READ', NDF2, STATUS )
      CALL NDF_MBND( 'TRIM', NDF1, NDF2, STATUS )             [6]

*  Create a new output NDF based on the first input NDF.
      CALL NDF_PROP( NDF1, 'Axis,Quality', 'OUT', NDF3, STATUS )  [7]

*  Map the input and output data arrays.
```

[3]The symbolic constant SAI__OK is defined in the include file SAE_PAR.

```
      CALL NDF_MAP( NDF1, 'Data', '_REAL', 'READ', PNTR1, EL, STATUS )  [8]
      CALL NDF_MAP( NDF2, 'Data', '_REAL', 'READ', PNTR2, EL, STATUS )
      CALL NDF_MAP( NDF3, 'Data', '_REAL', 'WRITE', PNTR3, EL, STATUS )

*  Check that the input arrays do not contain bad pixels.
      CALL NDF_MBAD( .FALSE., NDF1, NDF2, 'Data', .TRUE., BAD, STATUS )  [9]

*  Add the data arrays.
      CALL ADDIT( EL, %VAL( PNTR1( 1 ) ), %VAL( PNTR2( 1 ) ),  [10]
     :            %VAL( PNTR3( 1 ) ), STATUS )

*  End the NDF context.
      CALL NDF_END( STATUS )                                   [12]
      END


*  Subroutine to perform the addition.
      SUBROUTINE ADDIT( EL, A, B, C, STATUS )                  [11]
      INCLUDE 'SAE_PAR'
      INTEGER EL, STATUS, I
      REAL A( EL ), B( EL ), C( EL )

      IF ( STATUS .NE. SAI__OK ) RETURN

      DO 1 I = 1, EL
         C( I ) = A( I ) + B( I )
 1    CONTINUE
      END
```

Programming notes:

(1) Note that the application is actually a subroutine, called ADD, with a single integer argument called STATUS. This is the ADAM method of writing applications (see SUN/101).

(2) The INCLUDE statement is used to define standard "symbolic constants", such as the value SAI__OK which is used in this routine. Such constants should always be defined in this way rather than by using actual numerical values. The file SAE_PAR is almost always needed, and should be included as standard in every application.

(3) The value of the STATUS argument is checked. This is because the application uses the error handling strategy described in SUN/104, which requires that a subroutine should do nothing unless its STATUS argument is set to the value SAI__OK on entry. Here, we simply return without action if STATUS has the wrong value.

(4) An NDF *context* is now opened, by calling NDF_BEGIN. This call matches the corresponding NDF_END call at the end of the ADD routine. When the NDF_END call is reached, the NDF_ system will "clean up" by closing down everything which has been used since the matching call to NDF_BEGIN (see §3.4). Since we want to clean up everything in the application at this point, the initial call to NDF_BEGIN is put right at the start.

(5) The two input NDFs which we want to add are now obtained using the parameters 'IN1' and 'IN2'. Lots of things happen behind the scenes at this point, possibly involving

prompting the user to supply the names of the data structures to be added, and a pair of integer values NDF1 and NDF2 are returned. These values are *NDF identifiers* and are used to refer to the NDFs throughout the rest of the application (see §3.2).

(6) The first thing we do with these identifiers is to pass them to NDF_MBND. This routine ensures that the two NDFs are the same shape and size, which is what we require. The details of how this is done are explained much later (in §17.3). For now, just accept that it works.

(7) An output NDF is created next by calling NDF_PROP which uses the parameter 'OUT' to get the new data structure (probably prompting the user for its name) and returns another identifier for it in NDF3. NDF_PROP bases the new NDF on the first input NDF (see §14.4). This ensures that it's the right size, *etc.*, and also that it contains any ancillary information which can legitimately be copied from the input.

(8) The *data* arrays in the input and output NDFs are then accessed by calling NDF_MAP. Rather than returning actual data values, this routine returns *pointers* for the *data* values in PNTR1, PNTR2 and PNTR3 (see §8.2). Note that we want to 'READ' the input arrays and 'WRITE' to the output array.

(9) Since this is a very simple application, it cannot handle the special *bad*-pixel values which may be present in some NDF data structures. The call to NDF_MBAD at this point checks that there are none present (see §17.2). If there are, then an appropriate error message will result and the application will abort.

(10) The subroutine ADDIT which performs the work is now called. The pointer values returned by NDF_MAP are turned into actual Fortran arrays at this point, which ADDIT can access. This is done by using the %VAL function in the call to ADDIT (see §8.2).

(11) ADDIT itself is a very simple subroutine. Since all the arrays it will be passed are the same size (we have ensured this), there is no need to worry about their dimensions. They are all handled as if they were 1-dimensional, and simply added. The application could easily be altered to perform a different function by changing this routine.

(12) Finally, NDF_END is called. As already explained, this shuts everything down, ensuring that all NDF data files are closed, *etc.* before the application finishes.

3 OBTAINING AND USING NDF IDENTIFIERS

Having given an overview of the NDF_ system, the mechanism by which NDF data structures are accessed by an application and subsequently referred to by means of *identifiers* is now discussed in detail.

3.1 Accessing NDFs for Input

The first task to perform before any processing of NDF data can take place is to gain access to an NDF data structure. The creation of new NDFs (*e.g.* to contain output from an application) is left until later (see §13.2), when some of the concepts involved should be clearer. To start with, we

assume that an NDF structure exists and that we are going to write an application which needs to access it.

The normal method of obtaining access to an NDF for input is via a *parameter*, using the routine NDF_ASSOC which *associates* an NDF with the parameter:

```
INTEGER INDF

...

CALL NDF_ASSOC( 'IN', 'READ', INDF, STATUS )
```

Here, the 'IN' is being used to obtain 'READ' access to an NDF data structure. This means that we will be able to read values from the NDF, but not to modify it. An integer variable INDF is supplied to receive the returned NDF identifier value.

The effect of this call to NDF_ASSOC is that the programming environment will attempt to find a suitable pre-existing NDF data structure, most probably by prompting the person running the application to type in its name. If you are using ADAM, you have considerable control over exactly how the data structure is obtained, but none of this need be specified in the application. Instead, a separate *interface file* (with a file type of .ifl) is used for this purpose. Interface files are discussed in SUN/101 and full details can be found in the reference document SUN/115. Here, only a very simple interface file is given as an example:

```
interface PROG              # The name of the application

   parameter IN             # The name of the parameter
      prompt   'Input NDF data structure'
   endparameter

endinterface
```

Other parameter entries may also be present in the same file (the text beginning with the '#' signs on the right is commentary and need not be included). In combination with the call to NDF_ASSOC, this interface file would result in a prompt, to which the user could respond with the name of an NDF data structure, most probably simply the name of an HDS container file, as follows:

```
IN - Input NDF data structure > datafile
```

Assuming that the data structure is a valid NDF, NDF_ASSOC will return an identifier for it via its INDF argument and this may then be passed to other NDF_ routines to access the NDF's values. If the data structure is not valid, then an error message will appear and the prompt will be repeated until a valid response is given or the user decides to give up (in ADAM this would be done by typing '!!' – the *abort* response). In this latter case, NDF_ASSOC will return with its STATUS argument set to an error value and INDF will be set to the value of the symbolic constant NDF__NOID, a universal value (defined in the include file NDF_PAR) which indicates that an NDF identifier is not valid.

Note that the routine NDF_EXIST may also be used to access NDF data structures for input, but is more typically used during the creation of new NDFs. Its use is described later in §13.3.

3.2 NDF Identifiers

As has been illustrated above, the NDF_ routines refer to NDFs by means of values held in integer variables called *NDF identifiers*. Of course, these integers are not NDF data structures themselves; they simply identify the data structures, the internal details of which are hidden within the NDF_ system.

Each NDF identifier has a unique value which will not be re-used, and this property makes it possible to tell at any time whether an integer value is a valid NDF identifier or not. An identifier's validity depends on a number of things, such as its actual value (the value NDF__NOID is never valid for instance) and the previous history and current state of the NDF_ system (an identifier which refers to a data structure which has been deleted will no longer be valid). Note that identifier values should not be explicitly altered by applications, as this may also cause them to become invalid.

Identifier validity can be determined by using the routine NDF_VALID, which returns a logical value of .TRUE. via its VALID argument if the identifier supplied is valid:

```
CALL NDF_VALID( INDF, VALID, STATUS )
```

This is the only NDF_ routine to which an invalid identifier may be passed without provoking an error.

3.3 Annulling Identifiers

The number of NDF identifiers available at any time is quite large but is, nevertheless, limited. Each identifier also consumes various computer resources which may themselves be limited, so it is important to ensure that identifiers are *annulled* once they have been finished with, *i.e.* when access to the associated NDF data structure is no longer required. This is particularly important when an application is to be run from a programming environment where the same executable program continues to run indefinitely (*e.g.* when using the ADAM command language ICL). Failure to annul every NDF identifier before the application finishes can result in the program eventually being unable to continue, an event which may be accompanied by strange error messages.

Annulling an NDF identifier renders it invalid and resets its value to NDF__NOID. It differs from simply setting the identifier's variable to this value, however, because it ensures that all resources associated with it are released and made available for re-use. An identifier is annulled using the routine NDF_ANNUL, as follows:

```
CALL NDF_ANNUL( INDF, STATUS )
```

Note that annulling an invalid identifier will produce an error, but this will be ignored if the STATUS argument is not set to SAI__OK when NDF_ANNUL is called (*i.e.* indicating that a previous error has occurred). This means that it is not usually necessary to check whether an identifier is valid before annulling it, so long as the only possible reason for it being invalid is a previous error which has set a STATUS value.

3.4 NDF Contexts: BEGIN and END

Unfortunately, it is all too easy to neglect to annul an identifier, especially if the number of NDFs used by an application is large. It is a particular problem if an application is fairly complex and can potentially terminate at many points with an unknown number of NDF identifiers allocated. In such cases, it is often far easier simply to say "annul all the identifiers I've used in this part of the program". The NDF_BEGIN and NDF_END routines allow this.

These routines are used in pairs to delimit a section of an application, rather like a Fortran 77 IF...END IF block, although often they will enclose the entire application. When NDF_END is called, it will annul all the NDF identifiers issued since the matching call to NDF_BEGIN, for example:

```
CALL NDF_BEGIN
CALL NDF_ASSOC( 'IN1', ... )
CALL NDF_ASSOC( 'IN2', ... )

<allocate more NDF identifiers>

CALL NDF_END( STATUS )
```

As with IF...END IF blocks, matching pairs of calls to NDF_BEGIN and NDF_END may be nested, every call to NDF_END being accompanied by a corresponding call to NDF_BEGIN. This makes it possible to annul only those identifiers acquired within a certain part of an application if required. So long as no NDF identifiers are acquired outside the outermost BEGIN...END block, this type of program structure provides a complete safeguard against forgetting to annul an identifier. Explicit calls to NDF_ANNUL can then largely be eliminated.

3.5 Cloning Identifiers

Since an NDF identifier only refers to an NDF data structure and is not itself a data structure, it is possible to have several identifiers referring to the same NDF. A duplicate identifier for an NDF may be derived from an existing one by a process called *cloning*, which is performed by the routine NDF_CLONE, as follows:

```
CALL NDF_CLONE( INDF1, INDF2, STATUS )
```

This returns a second identifier INDF2 which refers to the same NDF data structure as INDF1.

Cloning is not required frequently, but it can occasionally be useful in allowing an application to "hold on" to an NDF when an identifier is passed to a routine which will annul it; *i.e.* you simply pass the original identifier and keep the cloned copy. An example of this can be found in the section on matching NDF attributes (§17.3).

4 NDF SHAPE AND SIZE INFORMATION

This section describes the attributes which determine an NDF's shape and size, and shows how the values of these attributes can be obtained and used.

4.1 Dimensionality and Bounds

An NDF's *shape* is determined by its *dimensionality* (*i.e.* its number of dimensions) and the lower and upper bound of each dimension. In this respect, an NDF is exactly like a Fortran 77 array. It may have between 1 and 7 dimensions, each indexed by an integer lying between the lower and upper *pixel-index bounds* of that dimension. These bounds may take any integer values, so long as the lower bound does not exceed the upper bound. Each element of the array is termed a *pixel*.

All the array components of an NDF (*i.e.* its *data*, *variance* and *quality* components) have the same shape, so an NDF may either be regarded as a set of up to three matching N-dimensional arrays, or as an N-dimensional array of pixels each of which has a *data* (and optionally a *variance* and *quality*) value associated with it. In general, other non-array components may also be present, of course.

The usual notation can be used to refer to an NDF's shape. For instance:

(0:255, -1:1, 3)

describes the shape of a 3-dimensional NDF with pixel indices ranging from 0 to 255 in the first dimension, from -1 to 1 in the second dimension, and from 1 to 3 in the third dimension. The lower bound is taken to be 1 if not specified.

An NDF's pixel-index bounds and dimensionality may be determined from its identifier with the routine NDF_BOUND, as follows:

```
INTEGER NDIMX, LBND( NDIMX ), UBND( NDIMX ), NDIM

...

CALL NDF_BOUND( INDF, NDIMX, LBND, UBND, NDIM, STATUS )
```

Here, the integer arrays LBND and UBND are supplied to receive the lower and upper bounds of each dimension, and the number of dimensions is returned via the NDIM argument. The size of the LBND and UBND arrays is specified by the NDIMX argument and should normally be sufficiently large to ensure that all the NDF's bounds can be accommodated. The symbolic constant NDF__MXDIM (defined in the include file NDF_PAR) specifies the maximum number of dimensions which an NDF may have, and is often used when declaring the size of such arrays.

If the value of NDIMX is larger than the actual number of NDF dimensions, then any unused elements in the LBND and UBND arrays will be filled with the value 1. This allows an application to "pretend" that an NDF has more dimensions than it actually has, if that is convenient. For instance, suppose an application expects a 2-dimensional NDF and determines the pixel-index bounds as follows:

```
INTEGER LBND( 2 ), UBND( 2 ), NDIM

...

CALL NDF_BOUND( INDF, 2, LBND, UBND, NDIM, STATUS )
```

```
NDIM = 2

...
```

Although this application simply asserts that the NDF is 2-dimensional, it will nevertheless be able to handle 1-dimensional NDFs as well, because they will appear to the rest of the application as if their second dimension has pixel-index bounds (1:1).

Sometimes, this deception can also be practised if the NDF has more dimensions than expected. In this case, NDF_BOUND will return the expected number of pixel-index bounds and will discard the bounds information for any extra dimensions which are non-significant (*i.e.* for which the lower and upper bounds are equal). In the above example, the NDF would therefore appear 2-dimensional to the rest of the application. However, if any of the extra dimensions are significant (*i.e.* have unequal lower and upper bounds), then NDF_BOUND will report an error explaining the problem and return with its STATUS argument set to NDF__XSDIM (too many dimensions), as defined in the include file NDF_ERR. In the normal course of events, the application would then terminate and the error message would be delivered to the user.

The above simple program fragment will therefore ensure that an NDF is made to appear 2-dimensional if at all possible, and will correctly handle any difficulties which may arise.

4.2 Dimension Sizes

In many applications, knowledge of the pixel-index bounds of an NDF may be unnecessary and it is sufficient to know the dimension sizes (*i.e.* how many pixels there are along each dimension). This will normally be the case when converting an application that was originally written for a different data system, where typically the dimension size is the only shape information used.

The dimension size D is related to the lower and upper bounds of each dimension, L and U, by:

$$D = U - L + 1$$

but may be obtained more directly using the routine NDF_DIM, as follows:

```
INTEGER NDIMX, DIM( NDIMX ), NDIM

...

CALL NDF_DIM( INDF, NDIMX, DIM, NDIM, STATUS )
```

As with NDF_BOUND, an integer array DIM is provided to receive the returned dimension sizes, and any unused elements in this array will be assigned the value 1. If the array size NDIMX is less than the actual number of NDF dimensions, then information for extra non-significant dimensions (*i.e.* dimensions with sizes equal to 1) will be discarded. However, if any of the discarded dimensions would be significant (*i.e.* have a size greater than 1), then an error will result.

This again allows an application to make an NDF appear to have the number of dimensions it requires if at all possible. For instance, a smoothing application might expect an NDF to contain a 2-dimensional image, and could obtain the dimension sizes it requires, NX and NY, as follows:

```
INTEGER DIM( 2 ), NDIM, NX, NY

...

CALL NDF_DIM( INDF, 2, DIM, NDIM, STATUS )
NX = DIM( 1 )
NY = DIM( 2 )

...
```

No explicit check on the number of NDF dimensions would be necessary.

4.3 NDF Size

The *size* of an NDF is the total number of pixels lying within its bounds, and is given by the product of the sizes of all its dimensions. Many applications require only this information about an NDF's shape and can conveniently obtain it using the routine NDF_SIZE, as follows:

```
CALL NDF_SIZE( INDF, NPIX, STATUS )
```

This routine returns the total number of pixels via its integer argument NPIX. Note, however, that this same quantity is also returned as a by-product of most routines which directly access the array components of an NDF (see §8.2, for example), so a call to NDF_SIZE can often be avoided.

4.4 "Safe" Dimension Sizes under Error Conditions

A common pitfall which can affect software using the inherited status error-handling strategy employed by all the NDF_ routines (see §2.3) arises from the way in which the Fortran 77 "adjustable array" facility behaves.

Consider the following example, where a call to the NDF_SIZE routine is used to obtain the number of pixels in an NDF, and the returned result is then passed to another routine where it specifies the size of an adjustable array, A:

```
CALL NDF_SIZE( INDF, NPIX, STATUS )
CALL DOIT( NPIX, A, STATUS )
```

Normally, this will cause no problems. However, if there has been a previous error so that STATUS is no longer set to SAI__OK, then NDF_SIZE will not execute. If no further precautions were taken, the returned value of NPIX would be undefined and could easily have a value (zero, perhaps) which is invalid as an adjustable array dimension. In such cases, the application would probably terminate (*i.e.* crash) with a run-time error.

To prevent this happening, all NDF_ routines which return array size or dimensionality information will provide a "safe" value of 1 under error conditions. This applies both to the case where STATUS is set before the routine is called, as well as to the case where the NDF_ routine itself fails. This generally means that no extra precautions need to be taken to prevent run-time errors from this cause.

5 GENERAL PROPERTIES OF NDF COMPONENTS

This section describes how to refer to the components of an NDF and outlines the general concept of a component's *state*.

5.1 Specifying Component Names

Many NDF_ routines are capable of operating on more than one NDF component and use a character string holding one of the component names given in §2.1 to identify the component to be used. If required, these names may be abbreviated (to no less than 3 characters) and may be written using both upper- and lower-case characters. Thus:

'DATA', 'Data', 'Variance', 'Var', 'UNIT' *and* 'hist'

could all be passed to NDF_ routines as valid component names. To avoid confusion between component names and other character strings which may occasionally have the same value, all component names in programming examples in this document are given with a capitalised first letter followed by lower-case, *e.g.* 'Quality'.

Some NDF_ routines will also accept a list of components, in which the component names should be separated by commas, for instance:

'Data,Variance,Title'

Spaces may be included before and after component names, but may not be embedded within them.

5.2 Component State

Although the *state* of an NDF component is strictly an attribute of each individual component, it is nevertheless convenient to discuss it here because all components behave similarly in this respect, and the descriptions of component access which follow depend on understanding it.

A component's state is a logical value which indicates whether or not that component has a value or is available for use. If a component's state is .TRUE., then the component is said to be *defined* or in a *defined state*, whereas if its state is .FALSE., it is *undefined* or in an *undefined state*.[4] The state of any NDF component can be determined from its identifier by specifying the component name in a call to the routine NDF_STATE. For example:

```
LOGICAL STATE

...

CALL NDF_STATE( INDF, 'Variance', STATE, STATUS )
```

[4] A component's state as defined by the NDF_ system should not be confused with the state of data objects as defined by the underlying data system HDS. An "HDS-undefined" primitive object is normally regarded as illegal by the NDF_ system and provokes an error. The "NDF-state" of a component is instead related to the existence or non-existence of HDS objects, although not always in an obvious fashion.

will return a .TRUE. value if the NDF's *variance* component is defined, *i.e.* if there are variance values available in the NDF which could be read into an application and used if required. An attempt to read values from an undefined component can result in an error, although the NDF_ system may take other appropriate action in some circumstances, depending on the component.

In common with most NDF_ routines which take a component name as an input argument, a list of components may also be supplied to NDF_STATE, for example:

```
CALL NDF_STATE( INDF, 'Dat,Var,Lab', STATE, STATUS )
```

In this case, the routine will return the logical "AND" of the values which would have been obtained for each component alone, so in this example a .TRUE. value will be returned via the STATE argument only if all three of the *data*, *variance* and *label* components are available.

5.3 Factors Determining a Component's State

When an NDF is first created, all its components start out in an undefined state and they become defined only when values have been assigned to them. Note that with the exception of the *history* component (see §22.3), there is no separate concept of component "creation" in the NDF_ system – *i.e.* there is no need to "create" a component before assigning a value to it, and a component cannot "exist" without also having a value. The method by which values are assigned depends on the individual component, and is discussed in later sections.

The opposite process, *i.e.* of resetting a component to become undefined, is more straightforward and is the same for all components. It is performed by the routine NDF_RESET, which takes the name of a component or a list of components in the same way as NDF_STATE. For example:

```
CALL NDF_RESET( INDF, 'Data,Variance,Units', STATUS )
```

would reset the state of the three named components, making their values unavailable and therefore effectively erasing them.

5.4 Restrictions on the Data Component's State

So long as an NDF data structure is accessible to the NDF_ system (*i.e.* so long as at least one valid NDF identifier still refers to it) the state of its components can be manipulated freely by assigning values and making calls to NDF_RESET. However, when the final identifier referring to an NDF is annulled, that NDF is *released* from the NDF_ system. At this point, the resulting data structure must be a valid NDF if it is subsequently to be accessed by other applications.

This requires that at least the *data* component should be in a defined state at this point (all other NDF components are optional, but the *data* component must be present if the data structure as a whole is to be valid). An error will result if this is not the case, since this probably indicates a programming fault which has resulted in failure to assign values to the *data* component. However, this error will not result if the NDF_ system does not have the right to modify the data structure (*i.e.* if it was accessed only for reading) because the fault could not then have been in the current application.

6 ACCESSING CHARACTER COMPONENTS

The NDF *character components*, *title*, *label* and *units*, are all accessed through the same set of routines, which are described in this section.

6.1 Reading and Displaying Character Values

The routine NDF_CGET may be used to obtain the values of NDF character components. For instance:

```
CHARACTER * ( 80 ) VALUE

...

VALUE = 'Default label'
CALL NDF_CGET( INDF, 'Label', VALUE, STATUS )
```

will obtain the value of the *label* component, if defined, and return it via the VALUE argument. If the component is undefined, then no value will be returned so the default value established before the subroutine call would be used instead.

If the value of a character component is needed as part of a message, then it may be assigned directly to an MSG_ message token using the NDF_CMSG routine. Thus an application might generate a message about the *title* of an NDF as follows:

```
CALL NDF_CMSG( 'TITLE', INDF, 'Title', STATUS )
CALL MSG_OUT( 'PROG_TITLE', 'NDF title: ^TITLE', STATUS )
```

Here, 'TITLE' is the name of a message token (see SUN/104).

6.2 Writing Character Values

The routine NDF_CPUT may be used to assign new values to character components. For instance:

```
CALL NDF_CPUT( 'Surface Brightness', INDF, 'Lab', STATUS )
```

will assign the value 'Surface Brightness' to the *label* component, overwriting any previous value which this component had. Note that the entire character string (including trailing blanks if present) will be assigned, and the length of the NDF's character component will be adjusted to match the new value. After a successful call to NDF_CPUT, the character component's state becomes defined.

It is quite common for an application to obtain a new value for a character component via a parameter and then to store this value in an NDF. The routine NDF_CINP is therefore provided to do this directly. For instance, the following will obtain new values for all three character components via suitable parameters and write the values to an NDF:

```
CALL NDF_CINP( 'LABEL', INDF, 'Label', STATUS )
CALL NDF_CINP( 'TITLE', INDF, 'Title', STATUS )
CALL NDF_CINP( 'UNITS', INDF, 'Units', STATUS )
```

The first argument to NDF_CINP specifies the parameter to be used, while the third argument is the name of the NDF character component whose value is to be replaced. If a *null* parameter value is specified (by the user entering '!' in response to a prompt, for instance), then NDF_CINP will return without action, *i.e.* without setting a new value for the character component. A suitable default value for the component should therefore be established before NDF_CINP is called.

An example of an ADAM interface file parameter entry suitable for use with NDF_CINP can be found in §A.2.

6.3 Character Component Length

The length of an NDF character component (*i.e.* the number of characters it contains) is determined by the last assignment made to it, *e.g.* by NDF_CPUT, and may be obtained using the routine NDF_CLEN. For instance:

```
INTEGER LENGTH

...

CALL NDF_CLEN( INDF, 'Units', LENGTH, STATUS )
```

will return the number of characters in the *units* component via the LENGTH argument. If the specified component is undefined, then a value of zero will be returned.

6.4 Resetting Character Components

As described in §5.3, the value of a character component may be reset, thereby causing it to become undefined, by using the routine NDF_RESET. This routine will also accept a list of components. For instance:

```
CALL NDF_RESET( INDF, 'Title,Label,Units', STATUS )
```

will reset all three character components, effectively erasing them.

7 ARRAY COMPONENT TYPES

This section introduces the concept of the numerical *type*, or precision, of an NDF's array components and shows how the value of this attribute may be obtained and modified.

7.1 Numeric Types

Each of an NDF's array components (*i.e.* its *data*, *quality* and *variance*) has an attribute termed its *numeric type*, which refers to the numerical precision used to store its pixel values. The NDF_ system allows array components to be stored using any of seven numeric types which correspond with the seven primitive numeric types provided by the underlying data system HDS. It also uses the same character strings as HDS to refer to these types, as follows:

Character String	Numeric Type	Fortran Type
'_DOUBLE'	Double-precision	DOUBLE PRECISION
'_REAL'	Single-precision	REAL
'_INTEGER'	Integer	INTEGER
'_WORD'	Word	INTEGER * 2
'_UWORD'	Unsigned word	INTEGER * 2
'_BYTE'	Byte	BYTE
'_UBYTE'	Unsigned byte	BYTE

The HDS documentation (SUN/92) should be consulted for details of the range and precision of each of these numeric types, which may depend on the computer hardware in use. Note that character and logical types are not provided.

The numeric type of an array component is defined regardless of the component's state. The numeric type of the *data* component is specified when the NDF is created and the numeric type of its *variance* component will normally default to the same value, although both of these types may subsequently be altered (see §7.4). In contrast, the numeric type of the *quality* component is always '_UBYTE' and cannot be changed.

The numeric type of any NDF array component can be determined using the routine NDF_TYPE. For instance:

```
INCLUDE 'NDF_PAR'
CHARACTER * ( NDF__SZTYP ) TYPE

...

CALL NDF_TYPE( INDF, 'Data', TYPE, STATUS )
```

will return the numeric type of an NDF's *data* component as an upper case character string via the TYPE argument. Note how the symbolic constant NDF__SZTYP (defined in the include file NDF_PAR) should be used to declare the size of the character variable which is to receive the returned numeric type string.

The NDF_TYPE routine will also accept a list of array components. In this case, it will determine the numeric type of each component in the list and return the lowest-precision numeric type to which all these components can be converted without unnecessary loss of information. For instance, if values from the *data* and *variance* components of an NDF were to be combined, then a suitable numeric type for performing the processing could be obtained as follows:

```
CALL NDF_TYPE( INDF, 'Data,Variance', TYPE, STATUS )
```

If the *data* component held '_WORD' values and the *variance* component held '_REAL' values, then a value of '_REAL' would be returned indicating that both components could be accessed for processing using single-precision floating-point arithmetic without losing information. The question of type conversion during array component access is discussed later (see §8.7).

7.2 Complex Values

In addition to its numeric type, each NDF array component also has a logical *complex value flag* associated with it, which indicates whether it holds complex values. If so, then each pixel of that array is a complex number, with separate real and imaginary parts. Both parts share the same numeric type.

The complex value flag for an NDF's *data* component is established when the NDF is created and the *variance* component will normally adopt the same value by default, although both may subsequently be altered (see §7.4). In contrast, the *quality* component can never hold complex values, so its complex value flag remains set to .FALSE. and cannot be changed.

It is possible to determine whether an array component holds complex values by using the routine NDF_CMPLX. For instance:

```
CALL NDF_CMPLX( INDF, 'Variance', CMPLX, STATUS )
```

would return a logical .TRUE. result via the CMPLX argument if the NDF's *variance* component held complex values. As with NDF_TYPE, a list of components may also be specified, in which case the logical "OR" of the results for each component will be returned. Thus:

```
CALL NDF_CMPLX( INDF, 'Data,Variance', CMPLX, STATUS )
```

will return a .TRUE. result if either the *data* or *variance* component holds complex values.

7.3 Full Type Specifications

The combination of the seven numeric types and a complex value flag gives rise to a further seven complex types, which are conveniently expressed by prefixing the character string 'COMPLEX' to the numeric type, as follows:

```
'COMPLEX_DOUBLE'
'COMPLEX_REAL'
'COMPLEX_INTEGER'
'COMPLEX_WORD'
'COMPLEX_UWORD'
'COMPLEX_BYTE'
'COMPLEX_UBYTE'
```

Strings such as these, which represent both the numeric type and the complex value flag, are termed the *full type specification* of an NDF's array component. Note that a string such as '_REAL' can also be considered as a special case of a full type specification in which the "complex" field is blank.

The full type of an array component can be derived if necessary, but the routine NDF_FTYPE will return it directly. For instance:

```
INCLUDE 'NDF_PAR'
CHARACTER * ( NDF__SZFTP ) FTYPE

...

CALL NDF_FTYPE( INDF, 'Data', FTYPE, STATUS )
```

will return the full type of an NDF's *data* component as an upper-case character string via the FTYPE argument. Note how the symbolic constant NDF__SZFTP (defined in the include file NDF_PAR) should be used to declare the size of the character variable which is to receive the returned full type specification.

As might be expected, NDF_FTYPE will also accept a list of array components, returning a full type specification which combines the numeric types and the complex value flags of the individual components in the same way as the routines NDF_TYPE and NDF_CMPLX would have done individually. Thus:

```
CALL NDF_FTYPE( INDF, 'Data,Variance', FTYPE, STATUS )
```

will combine the full type specifications of both the *data* and *variance* components of an NDF, and return the resultant string.

7.4 Setting Component Types

So long as the necessary access is available (see §23.1), the full type of an NDF's *data* and *variance* components may be explicitly set when required, thereby altering the precision with which their values are stored. This may be done using the routine NDF_STYPE. For instance:

```
CALL NDF_STYPE( '_REAL', INDF, 'Data', STATUS )
```

would change the type of an NDF's *data* component to '_REAL'. This process will retain pixel values which may already be stored in the affected component; *i.e.* if these values are defined, then they will be converted to the new type and will not be lost. Of course, this may not always be needed, so it is possible to arrange for existing values to be disposed of, if necessary, in order to avoid the cost of converting the pixel values. This is done by calling NDF_RESET to reset the component's values to an undefined state prior to changing its type, thus:

```
CALL NDF_RESET( INDF, 'Data', STATUS )
CALL NDF_STYPE( '_REAL', INDF, 'Data', STATUS )
```

No attempt will then be made by NDF_STYPE to convert individual pixel values, so changing the type of an array component becomes a relatively inexpensive operation. Note that no harm is done by calling NDF_RESET in this way even if the component's state is already undefined.

NDF_STYPE will also accept a list of components and will set them all to the same type. For instance:

```
CALL NDF_STYPE( 'COMPLEX_DOUBLE', INDF, 'Data,Variance', STATUS )
```

would set both the *data* and *variance* components of an NDF to be of type 'COMPLEX_DOUBLE' (*i.e.* to have real and imaginary parts, both stored as double-precision numbers). Changes may be made freely between non-complex and complex types, and conversion of pixel values will be performed if these values are defined; either an imaginary part filled with zeros will be supplied, or the imaginary part will be discarded, as appropriate.

8 ACCESSING ARRAY COMPONENTS

This section explains how to access the array components of an NDF (*i.e.* its *data*, *quality* and *variance* components) in order to read values from them, or to write new values to them.

8.1 Overview of Mapped Access to Array Components

The NDF_ system provides access to the values in an NDF's array components by means of a technique known as *mapping*. In this, an array is created in a region of the computer's memory and a *pointer* to it is returned to the calling routine. Access to a selected NDF array component then takes place by means of the application reading and writing values to and from this array. For instance, if the component's values are to be read, then the array would be filled with the requested values, which the application can then access.

The array of values stored in memory is usually made to appear to an application as if it were a normal Fortran array (with an appropriate numeric type and shape) so that the NDF component's values may be accessed merely by referring to this *mapped array* in the normal Fortran way. If appropriate, modifications may also be made to the values in the array and these will be returned to the appropriate NDF component, updating it, when it is *unmapped* – a process which completes the transfer of values. At this point, the allocated memory is released and no further reference to the mapped values can be made unless the array is mapped again.

8.2 Mapping and Unmapping

The process of mapping one of an NDF's array components for access may be performed using the routine NDF_MAP. For instance:

```
INTEGER PNTR( 1 ), EL

...

CALL NDF_MAP( INDF, 'Data', '_REAL', 'READ', PNTR, EL, STATUS )
```

will map an NDF's *data* component as an array of '_REAL' (*i.e.* single-precision) values so that the calling routine can 'READ' them. NDF_MAP returns an integer pointer to the mapped values via its PNTR argument and a count of the number of array elements which were mapped via its EL argument. (PNTR is actually a 1-dimensional integer array, so the pointer value in this example will be returned in its first element.)

An application can access the mapped values as if they were a Fortran array of real numbers by passing the pointer to a subroutine. On computers which support it, this should be done using the %VAL facility (see below). Afterwards, access is relinquished by *unmapping* the NDF component using the routine NDF_UNMAP. The following illustrates the process:

```
      INTEGER PNTR( 1 ), EL
      REAL SUM

      ...

      CALL NDF_MAP( INDF, 'Data', '_REAL', 'READ', PNTR, EL, STATUS )
      CALL SUMIT( EL, %VAL( PNTR( 1 ) ), SUM )
      CALL NDF_UNMAP( INDF, 'Data', STATUS )

      ...

      END


      SUBROUTINE SUMIT( EL, ARRAY, SUM )
      INTEGER EL, I
      REAL ARRAY( EL ), SUM

      SUM = 0.0
      DO 1 I = 1, EL
         SUM = SUM + ARRAY( I )
 1    CONTINUE
      END
```

Here, the call to NDF_MAP maps the NDF's *data* component and returns a pointer to the mapped values. This pointer is passed to the subroutine SUMIT using %VAL, where it appears as a normal Fortran real array with EL elements.[5] SUMIT can then access the mapped values, in this case summing them, before it returns. Finally NDF_UNMAP is called to unmap the *data* component, thereby releasing the resources that were used.

In the above example, the array was treated as if it were 1-dimensional, since its actual dimensionality was not important. This is sometimes the case, but if the shape of the array is significant, then a call to a routine such as NDF_DIM would be used to obtain the necessary dimension size information, which would then be passed to the subroutine which accesses the mapped values, for instance:

```
      INTEGER PNTR( 1 ), EL, DIM( 2 ), NDIM
      REAL SUM
```

[5]Note that the NDF_MAP routine (together with NDF_AMAP, NDF_MAPQL and NDF_MAPZ which also obtain mapped access to arrays) will return a "safe" value of 1 via its EL argument under error conditions. This is intended to avoid possible run-time errors due to invalid adjustable array dimensions (see §4.4).

```
      ...

      CALL NDF_MAP( INDF, 'Data', '_REAL', 'READ', PNTR, EL, STATUS )

      CALL NDF_DIM( INDF, 2, DIM, NDIM, STATUS )
      CALL SUMIT( DIM( 1 ), DIM( 2 ), %VAL( PNTR( 1 ) ), SUM )

      CALL NDF_UNMAP( INDF, 'Data', STATUS )

      ...

      END
```

In this case, the routine SUMIT would declare the array to have the appropriate shape, and access it accordingly, as follows:

```
      SUBROUTINE SUMIT( NX, NY, ARRAY, SUM )
      INTEGER NX, NY, I
      REAL ARRAY( NX, NY ), SUM

      SUM = 0.0
      DO 2 J = 1, NY
         DO 1 I = 1, NX
            SUM = SUM + ARRAY( I, J )
 1       CONTINUE
 2    CONTINUE
      END
```

The routine NDF_MAP will also accept a list of component names and will map all of them in the same way (*i.e.* with the same type and mapping mode). Pointers to each mapped array will be returned via the PNTR argument, which must have sufficient elements to accommodate the returned values. The following shows how both the *data* and *variance* components of an NDF might be mapped for access, passed to a subroutine DOIT for processing, and then unmapped:

```
      INTEGER PNTR( 2 ), EL

      ...

      CALL NDF_MAP( INDF, 'Data,Variance', '_REAL', 'READ', PNTR, EL, STATUS )
      CALL DOIT( EL, %VAL( PNTR( 1 ) ), %VAL( PNTR( 2 ) ), STATUS )
      CALL NDF_UNMAP( INDF, 'Data,Variance', STATUS )

      ...
```

Note how NDF_UNMAP is also provided with a list of components to unmap. An alternative would be to specify a component name of '∗', which would cause NDF_UNMAP to unmap all components which had previously been mapped using the specified NDF identifier.

8.3 Implicit Unmapping

It is always necessary for an array component to be unmapped when access to it is complete. Failure to do this could result in an application running out of some resource, or failing to

re-access the component at a later time. In the case where values are being written to an NDF component (described below), failure to unmap that component afterwards may also result in the new values being lost.

However, the need to call NDF_UNMAP explicitly can often be avoided because several other routines will perform this operation implicitly if necessary. For instance, if NDF_ANNUL is used to annul an NDF identifier (see §3.3), then any components which were mapped through that identifier will first be implicitly unmapped. The effect is as if:

```
CALL NDF_UNMAP( INDF, '*', STATUS )
```

had been executed first.

Unmapping is also performed implicitly by the routine NDF_END as part of the "cleaning-up" service it provides (see §3.4). Any NDF identifiers which are implicitly annulled by NDF_END will also have any associated components unmapped at the same time. Careful use of the routines NDF_BEGIN and NDF_END can therefore eliminate many calls both to NDF_UNMAP and NDF_ANNUL.

8.4 Writing and Modifying Array Component Values

The *mapping modes* 'WRITE' and 'UPDATE' may be specified in calls to NDF_MAP in place of 'READ' to indicate that new values are to be written to an NDF's array component, or that its existing values are to be updated (*i.e.* modified), respectively. The mapped values can then be accessed in exactly the same was as for read access, except that any modifications made to the mapped values will be reflected in the actual values stored in the data structure. The transfer of these new values back to the NDF is completed when the array is unmapped, for instance:

```
CALL NDF_MAP( INDF, 'Variance', '_REAL', 'WRITE', PNTR, EL, STATUS )
CALL SETVAR( EL, %VAL( PNTR( 1 ) ), STATUS )
CALL NDF_UNMAP( INDF, 'Variance', STATUS )
```

Here, an NDF's *variance* component is mapped for 'WRITE' access and passed to a routine SETVAR which assigns values to it. When NDF_UNMAP is called, these new values are transferred to the NDF. If the *variance* component was previously in an undefined state, it now becomes defined.

8.5 More About Mapping Modes

The *mapping mode* argument of NDF_MAP is used to indicate how the transfer of mapped values should be handled when an array component is accessed. The three basic mapping modes 'READ', 'UPDATE' and 'WRITE' each has a separate use, and the following describes how they operate in detail:

'READ' – When an array component is mapped for 'READ' access, its pre-existing values are read into memory and made available to the calling routine (if the component's state is undefined, then an error will result since there will be no values to read). When the component is unmapped, the mapped values are discarded. If any have been modified they do not result in a permanent change to the data structure (in fact the values should not be modified, because the region of memory used may sometimes be write-protected and this will result in a run-time error).

'UPDATE' – When an array component is mapped for 'UPDATE' access, its pre-existing values are read into memory exactly as for 'READ' access (an error will similarly be produced if the component is in an undefined state). When the component is unmapped, any modifications made to the mapped values are written back to the NDF, which is therefore permanently modified.

'WRITE' – When an array component is mapped for 'WRITE' access, an array is created to receive new values, but no pre-existing values are read in. Since the mapped array is un-initialised it may contain garbage at this point. All elements of the array must therefore be assigned values by the application before the component is unmapped, at which point these values will be written to the data structure to become permanent. 'WRITE' access will succeed even if the component is initially in an undefined state, and the component's state will become defined once the unmapping operation has completed successfully.

8.6 Initialisation Options

Variations on the above basic mapping modes may be obtained by appending either of the two *initialisation options* '/BAD' or '/ZERO' to the mapping mode argument passed to NDF_MAP. These options specify the value to be supplied as "initialisation" in the absence of defined values. For instance, if 'READ/ZERO' is specified, thus:

```
CALL NDF_MAP( INDF, 'Data', '_REAL', 'READ/ZERO', PNTR, EL, STATUS )
```

then NDF_MAP will read the component's values if available, but instead of producing an error if its state is undefined, it will supply an array full of zeros instead. In similar circumstances, the access mode 'READ/BAD' would supply an array full of the *bad*-pixel value appropriate to the numeric type in use.[6] The same technique applied to update access (*e.g.* a mapping mode of 'UPDATE/BAD') has a similar effect, except that in this case the values, with possible subsequent modifications, will be written back to the array component when it is unmapped, causing its state to become defined.

The effect of initialisation options on write access (*e.g.* a mapping mode of 'WRITE/ZERO') is slightly different. In this case the initial state of the component is irrelevant, and any initialisation option is always used to initialise the array's values when it is first mapped. These values, with subsequent modifications, are then written back to the NDF component when it is unmapped. Note that there is generally no point in using an initialisation option with write access if an application will subsequently assign values to all of the array's pixels, but it may be useful if only certain pixels will subsequently be assigned values as it can save having to explicitly assign default values to all the others.

8.7 Implicit Type Conversion

An application seeking access to an NDF array component will not usually be concerned with the actual numeric type used to store the values and will request a type which suits whatever processing is to be performed. Single-precision (*i.e.* '_REAL') values are normally most appropriate and have been specified in previous examples.

[6] To understand this fully it may be necessary to read the later section on *bad pixels* (§9).

Nevertheless the third (TYPE) argument to NDF_MAP may be used to specify any of the seven numeric types supported by the NDF_ system (see §7.1) and a mapped array of values having the requested type will be made available. For instance:

```
CALL NDF_MAP( INDF, 'Data', '_WORD', 'READ', PNTR, EL, STATUS )
```

will map an array of '_WORD' (*i.e.* Fortran INTEGER∗2) values, performing type conversion automatically if required. If the component's values are being written or modified, then type conversion will also be performed, in reverse, when it is unmapped. The resulting "back-converted" values are used to update the NDF.

Unfortunately, type conversion takes time, so it may sometimes be worth avoiding if possible. This can be done, although only at the expense of some additional programming effort, by calling NDF_TYPE to determine the numeric type of a component which is to be mapped and using the same type for access, so that type conversion becomes unnecessary, for instance:

```
INCLUDE 'NDF_PAR'
CHARACTER * ( NDF__SZTYP ) TYPE

...

CALL NDF_TYPE( INDF, TYPE, STATUS )
CALL NDF_MAP( INDF, 'Variance', TYPE, 'READ', PNTR, EL, STATUS )
```

The application must then be prepared to explicitly process whatever numeric type is supplied, which will normally mean duplicating the main processing algorithm for each possible type. Such steps are not justified for normal use, but may sometimes be adopted for general-purpose software which will be heavily used. The GENERIC compiler (see SUN/7) can be useful for generating the necessary multiple copies of suitably-written algorithms if this approach is to be used.

8.8 Accessing Complex Values

Array components which hold complex values may be accessed using the routine NDF_MAPZ, which is identical to NDF_MAP except that it returns a pair of pointers for each component mapped; one for each of the real and imaginary parts. For instance:

```
INTEGER RPNTR( 1 ), IPNTR( 1 )

...

CALL NDF_MAPZ( INDF, 'Data', '_DOUBLE', RPNTR, IPNTR, EL, STATUS )
```

will return pointers to the real (*i.e.* non-imaginary) and imaginary parts of an NDF's *data* component, as double-precision values, via the arguments RPNTR and IPNTR. As with the routine NDF_MAP, a list of components may also be supplied and the RPNTR and IPNTR arguments must then have sufficient elements to accommodate the returned pointers. Unmapping of components mapped using NDF_MAPZ is performed in exactly the same manner as if NDF_MAP had been used.

Implicit type conversion can be performed between non-complex and complex types (and *vice versa*) if required, so the use of NDF_MAPZ is not restricted to array components which hold complex values. Equivalent comments also apply to NDF_MAP. A non-complex component accessed via NDF_MAPZ will be converted to have an imaginary part of zero, while a complex component accessed using NDF_MAP will have its imaginary part discarded.

8.9 Mapping the Variance Component as Standard Deviations

The values held in an NDF's *variance* component are estimates of the mean square error expected on the corresponding values stored in its *data* component. This particular method of storage is chosen because it allows more efficient computation in the common situation where errors from two or more sources are to be combined. For instance, errors are usually combined by adding the contribution from each source *in quadrature*, *i.e.* if σ_A and σ_B are the errors associated with two independent values, then the error σ in their sum is given by:

$$\sigma = \sqrt{\sigma_A^2 + \sigma_B^2}$$

The corresponding *variance* values therefore combine by simple addition:

$$V = V_A + V_B$$

and this is a considerably faster computation to perform.

Nevertheless, the *variance* values stored in an NDF are unsuitable for direct use for such purposes as plotting error bars on graphs. In this case the square root of each *variance* estimate must be taken, in order to obtain an estimate of the *standard deviation*, which is the value normally employed. This can be done explicitly, but it is also possible to instruct the NDF_MAP routine to provide an array of standard deviation values directly by specifying a special component name of 'Error'. For example:

```
CALL NDF_MAP( INDF, 'Error', '_REAL', 'READ', PNTR, EL, STATUS )
```

would return a pointer to a mapped array of single-precision standard deviation (or "error") values, obtained by taking the square root of the values stored in the NDF's *variance* component. These values should later be unmapped in the usual way, either implicitly, or by an explicit call to NDF_UNMAP, thus:

```
CALL NDF_UNMAP( INDF, 'Variance', STATUS )
```

If a mapping mode of 'UPDATE' or 'WRITE' had been specified, then the mapped values would be squared before being written back to the NDF's *variance* component. (Note a component name of 'Variance' should be specified when unmapping an array of standard deviation values; it is only during the mapping operation that the name 'Error' may be used.)

The same procedure may also be used when obtaining mapped access to complex values using the routine NDF_MAPZ (§8.8). In this case, the process of taking the square root (or of squaring during the subsequent unmapping operation) will be applied separately to the real (*i.e.* non-imaginary) and imaginary parts.

8.10 Restrictions on Mapped Access

Certain restrictions are imposed on mapped access to NDF components, the most important being that each array component may only be mapped for access once through any NDF identifier. Thus, two successive calls to NDF_MAP (or NDF_MAPZ) specifying the same identifier and array component will result in an error. Of course, the component may later be re-mapped if it has first been unmapped, either by an intervening call to NDF_UNMAP, or implicitly.

One way of avoiding this restriction is to *clone* a duplicate identifier using NDF_CLONE so that a component can be mapped twice, once through each identifier. However, a more fundamental restriction exists if a possible access conflict may occur. Update and write access to NDF components is always exclusive, so the NDF_ system will reject any attempt to access a component twice if one of the attempts involves update or write access, although the same component may be accessed any number of times (through different identifiers) for read access alone.

In addition, arrays stored in scaled or delta form (rather than primitive or simple form), are currently read-only. An error will be reported if an attempt is made to map a scaled or delta array component for update or write access. If you need to modify the values in such an array, then you should first make a copy of the NDF. Copying an NDF automatically converts all scaled amd delta arrays into simple arrays, and so the copied array can be mapped for any type of access.

9 BAD PIXELS

This section discusses the need to handle pixels whose values may be unknown, and describes the NDF facilities which support this concept.

9.1 The Need for Bad Pixels

The NDF_ system provides support for the concept of *bad* pixels (sometimes called “magic values” or “flagged values”) which may be present in the array components of an NDF. *Bad* pixels are pixels whose actual value may be unknown, and which have therefore been assigned a special “null” or *bad* value. They can arise in a wide variety of ways, for instance:

- In an image-processing algorithm, arithmetic problems, such as attempts to divide by zero, might prevent values being calculated for certain output pixels. These missing pixels could be designated as *bad*.
- In an interactive editing application, deleted pixel values might be denoted as *bad*, indicating that they no longer have a useful value.
- *Bad* pixels might be introduced into array components during the process of type conversion (*e.g.* using the routine NDF_STYPE – see §7.4) if values are encountered which cannot be represented using the component’s new type. The same situation can also arise if type conversion takes place implicitly when an array component is accessed by mapping and unmapping it (see §8.7).

- If the shape of an NDF is altered (see §21.1), then new pixels may be introduced by changes in its pixel-index bounds. Such pixels will usually be marked as *bad*, since they will never have had a value assigned to them.

Many other examples exist where pixel values cannot be meaningfully assigned, and the *bad pixel* concept is designed to handle this general problem.

9.2 Recognition and Processing of Bad Pixels

Having been introduced into an array, *bad* pixels must be recognisable by subsequent algorithms, which must be capable of taking suitable action to allow for the missing pixel values (normally, this will mean disregarding them in an appropriate manner).

For purposes of recognition, *bad* pixels are assigned a unique value, appropriate to their numeric type and reserved for this purpose. This value is identified by a symbolic name of the form VAL__BADx, where "x" identifies the numeric type, as follows:

VAL__BADD — *Bad double-precision pixel value*
VAL__BADR — *Bad single-precision pixel value*
VAL__BADI — *Bad integer pixel value*
VAL__BADW — *Bad word pixel value*
VAL__BADUW — *Bad unsigned word pixel value*
VAL__BADB — *Bad byte pixel value*
VAL__BADUB — *Bad unsigned byte pixel value*

(note the use of a double underscore character in this naming convention). These symbolic constants, along with others relating to the HDS primitive numeric types, are defined in the include file PRM_PAR (see SUN/39). The appropriate symbolic name should be used for the numeric type being processed.

To give an example of how the need for *bad* pixels inevitably arises in practice, consider the following simple algorithm to divide one single-precision (*i.e.* '_REAL') array A by another B, to give a result in the array C:

```
      INTEGER I, EL
      REAL A( EL ), B( EL ), C( EL )

      ...

      DO 1 I = 1, EL
         C( I ) = A( I ) / B( I )
 1    CONTINUE
```

This algorithm is obviously flawed, because if any element of B is zero, then it will fail. However, we can test for this case, and assign *bad* pixels to affected elements of the output array as follows:

```
*  Define the VAL__BADx constants.
      INCLUDE 'PRM_PAR'

      ...

      DO 1 I = 1, EL

*  Test for division by zero and assign a bad value to the output array.
         IF ( B( I ) .EQ. 0.0 ) THEN
            C( I ) = VAL__BADR

*  Otherwise, calculate the result normally.
         ELSE
            C( I ) = A( I ) / B( I )
         END IF
 1    CONTINUE
```

Subsequent algorithms will then need to recognise these *bad* pixels and take appropriate action. Normally, if a *bad* pixel is encountered as input to an algorithm, it should automatically generate a corresponding *bad* pixel for output, a process known as *bad pixel propagation*. The above algorithm could therefore be adapted to recognise *bad* pixels in either of the input arrays (A and B) and to propagate them, as follows:

```
      INCLUDE 'PRM_PAR'

      ...

      DO 1 I = 1, EL

*  If either input pixel is bad, then so is the output pixel.
         IF ( A( I ) .EQ. VAL__BADR .OR. B( I ) .EQ. VAL__BADR ) THEN
            C( I ) = VAL__BADR

*  Check for division by zero.
         ELSE IF ( B( I ) .EQ. 0.0 ) THEN
            C( I ) = VAL__BADR

*  Otherwise, calculate the result normally.
         ELSE
            C( I ) = A( I ) / B( I )
         END IF
 1    CONTINUE
```

Different action may be required in other algorithms, but the process illustrated here is typical.

9.3 The Bad-Pixel Flag

The need to handle *bad* pixels inevitably adds to the amount of programming required when writing an application and can also adversely affect the execution time (although usually not as badly as might be feared). Nevertheless, most programmers recognise the need to handle *bad* pixels as an unfortunate fact of life that often has to be catered for.

When possible, however, it can be useful to determine whether or not *bad* pixels are actually present in an array, so that a more streamlined algorithm can be used if there is no need to check each pixel explicitly for a *bad* value. Indeed, in some cases, an algorithm may depend upon the absence of *bad* pixels, so a method of checking for this condition is required. Each array component of an NDF therefore has a logical value associated with it called its *bad pixel flag*, which is intended to indicate whether or not *bad* pixels may be present in that component.

Unfortunately, certainty in this matter comes at a price, because there are many operations (including those performed by the NDF_ system) which might introduce *bad* pixels into an array component, but there is no way of being completely sure without performing additional checks. In many cases this will involve examining every array element, and the time taken to do this may mean that the test is hardly worth performing if the resulting knowledge only results in a small saving of execution time in the main algorithm.

The NDF_ system takes a pragmatic approach to this problem, by allowing a slightly "fuzzy" interpretation of the *bad*-pixel flag's value, but providing the option to make it more precise (at additional cost) if required. The normal interpretation of the *bad*-pixel flag's value is therefore as follows:

.FALSE.	$\Rightarrow$	*There are definitely no bad pixels present*
.TRUE.	$\Rightarrow$	*There may be bad pixels present*

(See §9.5 for how this may be changed.) The NDF_ system goes to some lengths to keep track of *bad* pixels, but if any doubt exists it will cautiously set the *bad*-pixel flag to .TRUE.. Thus, a .FALSE. *bad*-pixel flag value expresses a definite fact, while a .TRUE. value only "suggests" the presence of *bad* pixels.[7]

9.4 Obtaining and Using the Bad-Pixel Flag

The *bad*-pixel flag value for an array component of an NDF may be obtained using the routine NDF_BAD. For instance:

```
LOGICAL BAD

...

CALL NDF_BAD( INDF, 'Data', .FALSE., BAD, STATUS )
```

will return the logical *bad*-pixel flag for an NDF's *data* component via the BAD argument. This value might then be used to control the choice of algorithm for processing the *data* values. The selected algorithm must be prepared to handle *bad* pixels unless a .FALSE. value of BAD has been returned, in which case further checks for *bad* pixels can be omitted.

For example, a very simple algorithm to add 1 to each element of an integer array IA might be written to adapt to the absence or possible presence of *bad* pixels as follows:

[7] Note that a *bad*-pixel flag value of .FALSE. does not ensure that the numerical value normally used to represent *bad* pixels will be absent, but it does indicate that such numbers are to be interpreted literally (*i.e.* as "good" values) and not as *bad* pixels.

```
      INCLUDE 'PRM_PAR'
      INTEGER EL, IA( EL ), I
      LOGICAL BAD

      ...

*  There are definitely no bad pixels present.
      IF ( .NOT. BAD ) THEN
         DO 1 I = 1, EL
            IA( I ) = IA( I ) + 1
 1       CONTINUE

*  There may be bad pixels, so check for them.
      ELSE
         DO 2 I = 1, EL
            IF ( IA( I ) .NE. VAL__BADI ) THEN
               IA( I ) = IA( I ) + 1
            END IF
 2       CONTINUE
      END IF
```

Note that this example is provided to illustrate the principle only, and does not imply that all applications should be capable of performing such processing. In fact, it is expected that many applications may choose not to support the processing of *bad* pixels at all, and this is considered in more detail in §9.5 and §17.2.

Like other routines, NDF_BAD will also accept a list of array components, in which case it will return the logical "OR" of the *bad*-pixel flag values for each component. For instance:

```
      CALL NDF_BAD( INDF, 'Data,Variance', .FALSE., BAD, STATUS )
```

would return a logical value indicating whether there may be *bad* pixels in either the *data* or *variance* component of an NDF.

Finally, note that the *quality* component of an NDF also has a *bad*-pixel flag, but that its value is currently always .FALSE.. This behaviour may change in future.

9.5 Requesting Explicit Checks for Bad Pixels

In the above examples, the third (CHECK) argument to NDF_BAD was set to .FALSE., indicating that no additional checks for *bad* pixels were to be performed. In this case, the normal interpretation of the returned *bad*-pixel flag value (as described in §9.3) applies. However, if this CHECK argument is set to .TRUE., then NDF_BAD will perform any additional checks necessary to determine whether *bad* pixels are actually present, for instance:

```
      CALL NDF_BAD( INDF, 'Data', .TRUE., BAD, STATUS )
```

This may involve explicitly examining every pixel, which could take a significant time to perform, although this will not always be necessary. If explicit checking is requested in this manner, the meaning of a .FALSE. *bad*-pixel flag value is unchanged, but the interpretation of a .TRUE. value changes to become:

.TRUE. ⇒ *There is definitely at least one bad pixel present*

Requesting an explicit check in this way can therefore convert an otherwise .TRUE. value for the *bad*-pixel flag into a .FALSE. value, but the opposite effect cannot occur.

The main value if this feature arises whenever there is a substantial cost involved in dealing with *bad* pixels within a processing algorithm. For instance, a considerably less efficient algorithm may be required to take account of *bad* pixels, or a suitable algorithm may not even exist (such as in a Fourier transform application). In this case, the extra cost of explicitly checking whether *bad* pixels are actually present will probably be worthwhile. In the following example, an application which cannot handle *bad* pixels checks whether there are any present, and aborts with an appropriate error message if there are:

```
      INCLUDE 'SAE_PAR'
      CALL NDF_BEGIN

      ...

      CALL NDF_BAD( INDF, 'Data,Variance', .TRUE., BAD, STATUS )
      IF ( BAD .AND. ( STATUS .EQ. SAI__OK ) ) THEN
         STATUS = SAI__ERROR
         CALL NDF_MSG( 'DATASTRUCT', INDF )
         CALL ERR_REP( 'FFT_NOBAD',
     :   'The NDF structure ^DATASTRUCT contains bad pixels which this ' //
     :   'application cannot handle.', STATUS )
         GO TO 99
      END IF

      ...

 99   CONTINUE
      CALL NDF_END( STATUS )
      END
```

Here, the cost of an explicit check is justified, because the cost of not performing the check (*i.e.* of the application having to abort) is even greater. Note that if the *bad*-pixel flag is .FALSE., then NDF_BAD will not need to check all the pixel values, so this approach does not hinder the normal case where *bad* pixels are flagged as absent. A simpler method of performing the same check is given in §17.2.

9.6 Setting the Bad-Pixel Flag

When an NDF is first created, all its components are in an undefined state and the *bad*-pixel flag values of the *data* and *variance* components are set to .TRUE.. They remain set to .TRUE. until evidence is presented that values have been assigned to either of these components which do not contain any *bad* pixels. This information can normally only come from the application which assigns the values, so the routine NDF_SBAD is provided to explicitly set the *bad*-pixel flag value when appropriate. For instance:

```
      CALL NDF_SBAD( .FALSE., INDF, 'Data', STATUS )
```

constitutes a declaration by the calling routine that the *data* component of an NDF does not currently contain any *bad* pixels. It is entirely the calling routine's responsibility to ensure that this declaration is accurate, since subsequent applications which access the NDF may otherwise be mislead. In practice, this means that it must either have generated the values itself or have checked them in some way to be sure of the absence of *bad* pixels.

The converse situation may also arise, where an application is forced to introduce *bad* pixels into an array component which previously did not contain any. In this case, it must also declare this fact if subsequent applications are not to be mislead. This time, the first (BAD) argument to NDF_BAD would be set to .TRUE.. For instance:

```
      CALL NDF_SBAD( .TRUE., INDF, 'Data,Variance', STATUS )
```

Here, a simultaneous declaration for both the *data* and *variance* components has been made by specifying both component names.

If an application is uncertain whether *bad* pixels may have been introduced or not, then it should always err on the side of caution and set the *bad*-pixel flag value to .TRUE.. This is not irreversible, since separate general-purpose applications will exist to allow a user to explicitly check for *bad* pixels in an NDF and to adjust the *bad*-pixel flag if required.

Finally, note that the *bad*-pixel flag for the *data* and *variance* components always reverts to .TRUE. if the component is reset to an undefined state, *e.g.* by a call to NDF_RESET. If a component is in this state, then NDF_SBAD will return without action and the *bad*-pixel flag will remain set to .TRUE..

9.7 Interaction with Mapping and Unmapping

An important process which can introduce *bad* pixels into an array component is that of accessing the component's values. This can occur for a number of reasons, but most obviously because type conversion may take place implicitly when an array component is mapped and unmapped (see §8.2). This conversion has the potential to fail if the value being converted cannot be represented using the new type. No error will result in such cases, but the affected pixels will be assigned the appropriate *bad*-pixel value.

In consequence, the effective *bad*-pixel flag value for an array component may change when that array is accessed by mapping it, and may change again when it is later unmapped. If type conversion errors occur when mapping or unmapping an array component which is being modified or written, then the *bad*-pixel flag will end up set to .TRUE. due to the resulting *bad* pixels being written back to the data structure.

What this means in practice is that, on input, the *bad*-pixel flag value should normally be checked after any mapping operation has been performed. Similarly, if the *bad*-pixel flag is to be changed using NDF_SBAD following the assignment of new values to an output array, then this should be done before the component is unmapped. See §9.9 for an example.

9.8 Interaction with Initialisation Options

Normally, there is no way for the NDF_ system to know what values have been assigned to an array component, so responsibility for setting the *bad*-pixel flag rests entirely with the application. One exception to this occurs, however, if an initialisation option of '/ZERO' is used and results

in an array full of zeros being mapped (see §8.6). In this case, the *bad*-pixel flag will be set to .FALSE. reflecting the resulting absence of *bad* pixels.

9.9 A Practical Template for Handling the Bad-Pixel Flag

The detailed discussion of the *bad*-pixel flag above will probably have obscured what needs to be done in practice when writing a simple application, so this section is intended to redress the balance. The overall message is that the NDF_ system will look after all the details. The only golden rule to be followed is:

If you alter the values in an array component, then use NDF_SBAD to update the bad-pixel flag before unmapping it.

The following program fragment shows what action is typically required:

```
      INTEGER PNTR1( 1 ), PNTR( 1 ), EL, NBAD
      LOGICAL BAD

      ...

*  Map the input array for reading and the output array for writing.
      CALL NDF_MAP( INDF1, 'Data', '_REAL', 'READ', PNTR1, EL, STATUS )
      CALL NDF_MAP( INDF2, 'Data', '_REAL', 'WRITE', PNTR2, EL, STATUS )

*  See if the mapped input array may contain bad pixels.
      CALL NDF_BAD( INDF1, 'Data', .FALSE., BAD, STATUS )

      ...

      <process the data, counting new bad pixels in NBAD>

      ...

*  See if there may be bad pixels in the output array and declare their
*  presence/absence.
      BAD = BAD .OR. ( NBAD .NE. 0 )
      CALL NDF_SBAD( BAD, INDF2, 'Data', STATUS )

*  Unmap the arrays.
      CALL NDF_UNMAP( INDF1, 'Data', STATUS )
      CALL NDF_UNMAP( INDF2, 'Data', STATUS )
```

In this example, a single input *data* array is being processed to produce a single output array and the application proceeds as follows:

(1) After mapping the arrays as required, it enquires whether the mapped input values may contain *bad* pixels.

(2) The main processing stage then takes place, during which additional *bad* pixels may be introduced. Some indication is needed of whether this has occurred; in this case a count of the number of new *bad* pixels is assumed to be returned in NBAD.

(3) The *bad*-pixel flag for the output array is updated, taking account of *bad* pixels which were present initially, and those introduced by the processing performed above.

(4) The arrays are then unmapped.

In cases where the processing algorithm provides no clear indication of whether *bad* pixels may have been introduced, the application would need to err on the side of caution and set the output *bad*-pixel flag to .TRUE..

10 THE QUALITY COMPONENT IN MORE DETAIL

10.1 Purpose of the Quality Component

The *quality* component of an NDF is provided to allow individual pixels to be flagged as having certain properties which applications may wish to take into account when processing them. In general, these properties are expected to be of a binary or logical character, such as membership of a set, rather than being quantitative, although combining several binary values to construct a simple numerical scale is not excluded.

To give a simple practical example, it might be useful to flag all the pixels in an image which are contaminated by defects in the detector system from which it originates. It might also be useful to classify each pixel as lying either in the "sky" background, or in the observed "object". Armed with this sort of information, it then becomes possible to perform operations like:

- Fit a surface to the sky signal, excluding contaminated pixels.
- Count the number of pixels in the object.
- Replace all the contaminated pixels by interpolation.

An NDF's *quality* component allows up to eight such binary conditions to be flagged, so numerous other applications and possibilities obviously exist.

If this appears too complicated for the sort of work you have in mind, then the good news is that the *quality* component can be almost entirely ignored by most applications and the default action of the NDF_ system will take care of it automatically. However, for applications which wish to exploit the possibilities that the *quality* component offers, a set of NDF_ routines is provided to access and control it explicitly. Their use is described here.

10.2 Accessing the Quality Array Directly

The main part of the *quality* component is an array containing an 8-bit (unsigned byte) value for each NDF pixel. Each of the bits in these pixels can be thought of as providing a 1-bit logical mask extending over all pixels and can be used to signify the presence or absence of a particular property at that location.

Thus, if bit 1 (value 1) indicated a contaminated pixel, and bit 2 (value 2) indicated membership of the set of "sky" pixels, then a value of 3 (*i.e.* 1+2) would indicate a contaminated sky pixel,

while a value of 0 would indicate an uncontaminated "object" pixel. With 8 bits available, *quality* values can range from 0 to 255.

An application may access the *quality* array directly by means of the routine NDF_MAP. For instance, the following call maps it for read access in exactly the same way that the NDF's *data* component might be accessed:

```
CALL NDF_MAP( INDF, 'Quality', '_UBYTE', 'READ', PNTR, EL STATUS )
```

Note that when accessing the *quality* array in this way a numeric type of '_UBYTE' must always be used. The outcome of using any other type is unspecified, although a meaning might be assigned to such an operation in future.

'WRITE' and 'UPDATE' access is also available. When an NDF is first created its *quality* component is in an undefined state. As with other components, it becomes defined once values have been written to it. Either of the initialisation options '/BAD' or '/ZERO' may also be appended to the mapping mode specification (see §8.6), where '/BAD' causes initialisation to the value 255 (all bits set) and '/ZERO' causes initialisation to zero (all bits clear).

10.3 The Bad-bits Mask

The significance attached to each bit of the *quality* values is arbitrary, and it is expected that specialised software packages which need to exploit the full capabilities of the *quality* component will make a particular choice of bit assignments and interpret these to influence the processing they perform, possibly in quite subtle ways. Such applications can never be truly general-purpose, however, because the number of possible bit assignments is unlimited, so agreement between writers of different software packages about how to interpret each bit cannot be achieved.

Nevertheless, some way of taking account of *quality* values in general-purpose software is very desirable. To allow this, an additional part of the *quality* component, termed the *bad-bits mask* is provided. This is a single 8-bit (unsigned byte) value, whose purpose is to allow the 8-bit *quality* value associated with each pixel to be converted into a single logical value, which can then be interpreted in the same way by all general-purpose software.

The value of the bad-bits mask for an NDF is obtained by calling the routine NDF_BB:

```
BYTE BADBIT

...

CALL NDF_BB( INDF, BADBIT, STATUS )
```

The returned result BADBIT is an *unsigned byte* number. If the NDF's *quality* component is undefined and/or a bad-bits value has not previously been set, then a value of zero is returned.

A new value for the bad-bits mask may be set with the routine NDF_SBB:

```
CALL NDF_SBB( BADBIT, INDF, STATUS )
```

By definition, the *quality* value of an individual pixel QUAL is converted into a logical value by forming the bit-wise "AND" with the bad-bits mask BADBIT and testing if the result is zero. In effect, this means that each bit which is set in BADBIT acts as a switch to activate detection of the corresponding bit in QUAL. If detection of a particular bit is activated and that bit is set in the QUAL value, then the corresponding NDF pixel is to be regarded as *bad*, and should be omitted from processing in much the same way as any pixel which has been explicitly assigned the *bad*-pixel value (see §9.4).

In Fortran implementations which support bit-wise operations, the logical value might be computed as follows:

```
OK = IIAND( IZEXT( QUAL ), IZEXT( BADBIT ) ) .EQ. 0
```

where a .FALSE. result indicates a *bad* pixel. However, this operation clearly requires non-standard Fortran functions and is potentially non-portable, so a function NDF_QMASK is provided to perform this task and to hide the implementation details. NDF_QMASK is a statement function and is defined by putting the following include statement into an application, normally immediately after any local variable declarations:[8]

```
INCLUDE 'NDF_FUNC'
```

Conversion of the *quality* value into a logical value is then performed by the function as follows:

```
OK = NDF_QMASK( QUAL, BADBIT )
```

The following example shows how a simple loop to find the average value of a pixel in an image array DATA might be written so as to take account of associated *quality* values, and exclude affected pixels:

```
BYTE QUAL( NX, NY )
INTEGER NGOOD, NX, NY, I, J
REAL DATA( NX, NY ), SUM, AVERAG
INCLUDE 'NDF_FUNC'

...

SUM = 0.0
NGOOD = 0
DO 2 J = 1, NY
   DO 1 I = 1, NX
      IF ( NDF_QMASK( QUAL( I, J ), BADBIT ) ) THEN
         SUM = SUM + DATA( I, J )
         NGOOD = NGOOD + 1
      END IF
```

[8]The use of a single include file to declare and define a statement function and its arguments is normally satisfactory. However, it sometimes makes it impossible to satisfy Fortran 77 statement order restrictions. For instance, if several such functions were defined in this way from separate subroutine libraries, it would not be possible to place all the declaration statements in front of all the function definition statements, as Fortran 77 requires. To cope with such problems, the single NDF_FUNC file may be replaced if necessary with the equivalent two separate include files NDF_FUNC_DEC and NDF_FUNC_DEF (in this order) which respectively declare and define the function.

```
1        CONTINUE
2     CONTINUE

      IF ( NGOOD .GT. 0 ) AVERAG = SUM / REAL( NGOOD )
```

As an alternative to the above explicit conversion of *quality* values into logical values, it is also possible to gain access to the *quality* array already converted into this form by calling the routine NDF_MAPQL, as follows:

```
      CALL NDF_MAPQL( INDF, PNTR, EL, BAD, STATUS )
```

This routine returns a pointer to a mapped array of logical values which are derived from the *quality* array and the current value of the bad-bits mask following the prescription above. Only read access is obtained by this routine and any modifications made to the mapped values will not result in a permanent change to the *quality* component. If the *quality* component is undefined, then an array of .TRUE. values is returned.

NDF_MAPQL also has an additional argument BAD, which indicates whether there are any .FALSE. values in the mapped array it returns. If BAD is returned .FALSE., then all the mapped values will have the value .TRUE., so *quality* information is either absent or, in combination with the bad-bits mask, has no effect. In such cases it may be possible to omit handling of *quality* values altogether and affect an improvement in algorithmic efficiency as a result.

Finally, it should be noted that while the use of NDF_MAPQL may appear more convenient than explicitly handling the *quality* values and bad-bits mask, it nevertheless involves an additional pass through the *quality* array and will therefore be a less efficient option.

10.4 Why Ignoring the Quality Component Works

Since the purpose of the *quality* component in general-purpose software is to indicate (in conjunction with the bad-bits mask) whether each pixel is "good" or *bad*, it is natural to ask whether *quality* component information can be handled in the same way as *bad* pixels. The answer is "yes". In fact the NDF_ routines assume by default that *quality* information will be handled in this way, and this makes it legitimate simply to ignore the presence of *quality* information in most cases.

What actually happens to make this possible is that by default all access to an NDF's *data* and *variance* components implicitly takes account of the possible presence of an associated *quality* component. For instance, if NDF_BAD is called to determine if *bad* pixels may be present in an NDF's *data* array (see §9.4), then an implicit check will also be made (if necessary) to see whether a *quality* component is also present. A further check on the bad-bits value may also be made, and all this information will be combined to determine whether *bad* pixels may be present.

When values are read from the *data* array (by mapping it for read or update access), a similar process takes place. In this case, if *quality* information is present, it is combined with the bad-bits value and used to insert *bad* pixels into the mapped *data* wherever the *quality* masking function evaluates to .FALSE.. The application accessing the *data* array need not know that this is happening, and can process the resulting mapped values as normal (but taking due account of the *bad*-pixel values). Note that no similar process takes place when values are written back to an NDF component (*i.e.* when any component accessed in update or write mode is unmapped); in this case the array values are transferred without interference from the *quality* component.

The same *automatic quality masking* also takes place by default when values are read from the *variance* component. This indicates why it may be more efficient to map both the *data* and *variance* components in a single call to NDF_MAP, e.g:

```
INTEGER PNTR( 2 ), EL

...

CALL NDF_MAP( INDF, 'Data,Variance', '_REAL', 'READ', PNTR, EL, STATUS )
```

which returns a pair of pointers in the PNTR array. By doing this, it becomes possible for NDF_MAP to access the *quality* array (if it exists) only once, and insert *bad* pixels into both mapped arrays simultaneously.

10.5 Controlling Automatic Quality Masking

The automatic *quality* masking action described above is controlled by a logical *quality masking flag* associated with each NDF identifier, and this flag can be manipulated if necessary to modify the default behaviour.

When an NDF identifier is first issued, its *quality* masking flag is set to .TRUE.. In this state, all access to the NDF's *data* and *variance* components via this identifier will implicitly take account of any *quality* information present, as described above. So long as no explicit access to the NDF's *quality* array is made, the *quality* masking flag remains set to .TRUE.. Thus, an application which chooses to completely ignore *quality* values will obtain the desired automatic *quality* masking action.

However, if explicit access to the *quality* array is obtained (*e.g.* by using the routines NDF_MAP or NDF_MAPQL), the flag will simultaneously be reset to .FALSE.. It is then assumed that the application intends to process *quality* values explicitly, so access to other NDF components will no longer take account of *quality* values. The *quality* masking flag returns to .TRUE. once direct access to the *quality* array is relinquished (by unmapping it using NDF_UNMAP). An application which intends to process *quality* values explicitly should therefore normally map the *quality* array first, so that subsequent access to other NDF components does not involve further implicit access to the *quality* component. The following sequence of operations might be typical:

```
CALL NDF_MAP( INDF, 'Quality', '_UBYTE', 'READ', QPNTR, EL, STATUS )
CALL NDF_MAP( INDF, 'Data', '_REAL', 'READ', DPNTR, EL, STATUS )

...

<process the data and quality information>

...

CALL NDF_UNMAP( INDF, 'Quality,Data', STATUS )
```

Here, the *quality* masking flag is set to .FALSE. by the first call to NDF_MAP, which accesses the *quality* component explicitly. The subsequent call to map the *data* component therefore ignores the *quality* values. Finally, the *quality* masking flag is returned to .TRUE. by the final call to NDF_UNMAP which unmaps the *quality* array.

The value of the *quality* masking flag associated with an NDF identifier may be determined at any time with the routine NDF_QMF:

```
LOGICAL QMF

...

CALL NDF_QMF( INDF, QMF, STATUS )
```

and a new value may be set with the routine NDF_SQMF:

```
CALL NDF_SQMF( QMF, INDF, STATUS )
```

This allows the normal behaviour to be over-ridden if necessary. For instance, if direct access to an NDF's *data* component is required without automatic *quality* masking occurring, then masking could be disabled as follows:

```
CALL NDF_SQMF( .FALSE., INDF, STATUS )
CALL NDF_MAP( INDF, 'Data', '_REAL', 'UPDATE', PNTR, EL, STATUS )
```

§A.8 contains an example of this technique in use.

11 EXTENSIONS

11.1 Extensibility

It will be evident from the rest of this document that the standard components of an NDF (*title*, *data*, *quality*, *etc.*) have rather well-defined meanings and rules associated with them which govern their interpretation and processing. This is necessary in order to allow items of general-purpose software to be written (often by independent authors) in such a way that they can be used together co-operatively and without the different authors having interpreted the meanings of the various components in different ways.

In contrast, *extensions* are the means by which the information stored in an NDF may be extended to suit local or personal requirements. For instance, an extension may be used to hold information specific to a particular form of processing, a particular software package, a particular author, or a particular astronomical instrument. In this case, universal agreement about how to process the information is not necessary, so no restrictions are imposed on the information which may be stored in extensions. As a consequence, a clear distinction has to be maintained between the processing of standard NDF components and of extensions, since truly general-purpose software can only be written to process the standard components of an NDF, whose interpretation is universally defined. If an NDF contains extension information, then this can only be successfully processed by more specialised software.

11.2 Extension Names and Software Packages

An NDF may contain any number of extensions which are distinguished by unique names. An extension name may consist of up to 15 characters. It must start with an alphabetic character, and may contain only alphanumeric characters (including underscore '_').

Extensions are normally associated with particular software packages which may use a particular extension name to identify additional information stored in an NDF in a form which only they understand.[9] For instance, a software package for analysing IRAS data might "own" the extension name 'IRAS' and use it to store additional information in NDF data structures relating to the processing of IRAS data. All the applications in that package would then be expected to recognise this extension in all NDF structures and correctly process the information it contains so that it remains valid throughout. However, there is no requirement for other software to be aware of this extension, other than to avoid using the same name.

In general, software which does not recognise an extension need do no more than *propagate* it (*i.e.* copy it), if appropriate, to any output data structure. Normally, this will be performed automatically by the NDF_ system and is considered in more detail in §14.

11.3 The Contents of Extensions

An NDF extension is an HDS object, normally a structure, whose contents are entirely at the discretion of the extension's designer. Although some simple NDF_ routines are described below for identifying and accessing extensions, the contents of extensions are generally unknown and their interpretation lies outside the scope of the NDF_ system. This information must therefore normally be handled by means of HDS routines, so anyone planning to use NDF extensions in their software will need to be familiar with the concepts used by HDS (see SUN/92).

In addition, some general guidelines aimed at minimising the risk of poor design and eliminating name clashes between extensions are to be found in SGP/38. These should be consulted by all potential designers of NDF extensions. The following additional recommendation should perhaps also be added:

Do not design over-elaborate extensions.

The freedom allowed by HDS is a great temptation to do so, but the cost of writing software to support the extension must always be kept in mind.

11.4 Accessing Existing Extensions

The existence or non-existence of a specified extension in an NDF can be determined using the routine NDF_XSTAT. For instance:

```
      CALL NDF_XSTAT( INDF, 'IRAS', THERE, STATUS )
```

[9] While the sharing of extensions between software packages and/or authors is not excluded, this is entirely the responsibility of those concerned. Since the requirements are so diverse, no specific recommendations can be made except to note that some documentary provision is normally necessary to ensure that separate authors interpret extension information in a consistent way.

will return a .TRUE. value via the logical argument THERE if an extension called 'IRAS' is present in an NDF. If it is, then it can be accessed using the routine NDF_XLOC. For instance:

```
CHARACTER * ( DAT__SZLOC ) LOC

...

CALL NDF_XLOC( INDF, 'IRAS', 'READ', LOC, STATUS )
```

will return an HDS locator to the extension (which is an HDS object) via the LOC argument.[10]

Note that an access mode of 'READ' was specified in this call to NDF_XLOC, indicating that the extension's contents would be read but not modified. An access mode of 'UPDATE' would be used if the contents were to be modified, while 'WRITE' access would only be used in order to *re*-write the extension's contents; in this case the NDF_ system will erase any previous contents before returning a locator to the extension.

The contents of an extension can be accessed by passing the resulting locator to suitable HDS routines. For instance, the value of an integer component called OFFSET within the 'IRAS' extension could be obtained as follows:

```
CHARACTER * ( DAT__SZLOC ) LOC
INTEGER OFFSET

...

CALL NDF_XLOC( INDF, 'IRAS', 'READ', LOC, STATUS )
CALL CMP_GET0I( LOC, 'OFFSET', OFFSET, STATUS )
CALL DAT_ANNUL( LOC, STATUS )
```

Note that the extension locator LOC must be annulled using the routine DAT_ANNUL when it is no longer required, since the NDF_ system will not perform this task itself.

This process of reading a scalar value from a component within an NDF extension is sufficiently common that a set of routines is provided to do it directly. These routines have names of the form NDF_XGT0x, where the (lower-case) "x" represents one of the standard Fortran 77 types and should be replaced by I, R, D, L or C, according to the type of value required. For instance, the above operation of reading an integer value from an 'IRAS' extension could also be performed with a single call to NDF_XGT0I, as follows:

```
CALL NDF_XGT0I( INDF, 'IRAS', 'OFFSET', OFFSET, STATUS )
```

These routines also have the advantage that they will accept a compound HDS component name as their third argument, making it possible to access components nested within sub-structures inside an extension. Array subscripts may also be specified, so that individual elements of non-scalar components and extensions may be accessed. For example:

```
CALL NDF_XGT0C( INDF, 'FITS', '(13)', RECORD, STATUS )
```

Would read the 13th element from a 'FITS' extension which consists of a 1-dimensional array of character strings.

[10]Note that an extension called 'IRAS' will actually be stored in an HDS object called .MORE.IRAS but the '.MORE.' should not be included when using NDF_ routines.

11.5 Creating New Extensions

New extensions are created by calling the routine NDF_XNEW and specifying the extension name together with its HDS type and shape. For instance:

```
CHARACTER * ( DAT__SZLOC ) LOC

...

CALL NDF_XNEW( INDF, 'IRAS', 'IRAS_EXTENSION', 0, 0, LOC, STATUS )
```

would create a new scalar 'IRAS' extension with an HDS type of 'IRAS_EXTENSION' and return a locator for it via the LOC argument. In practice, extensions will almost always be scalar HDS structures, but this routine allows for other possibilities if required. An error will result if the named extension already exists.

Once an extension structure has been created, components may be created within it and values assigned to them using HDS routines, as before. For instance, to create and assign a value of 30.5 to a real component called ANGLE in a newly-created 'IRAS' extension, the following calls might be used:

```
CHARACTER * ( DAT__SZLOC ) LOC
REAL ANGLE

...

CALL NDF_XNEW( INDF, 'IRAS', 'IRAS_EXTENSION', 0, 0, LOC, STATUS )
CALL CMP_MOD( LOC, 'ANGLE', '_REAL', 0, 0, STATUS )
CALL CMP_PUT0R( LOC, 'ANGLE', 30.5, STATUS )
CALL DAT_ANNUL( LOC, STATUS )
```

Here, the call to CMP_MOD ensures that the required ANGLE component exists, creating it if necessary. CMP_PUT0R then assigns a value to it. The extension locator LOC must be annulled when it is no longer required, since the NDF_ system will not perform this task.

Again, this process of creating a scalar component within an NDF extension and writing a value to it is sufficiently common that a set of routines is provided to do it directly. In this case, the routines have names of the form NDF_XPT0x, where the (lower-case) "x" should be replaced by I, R, D, L or C, according to the type of value being written. The above operation of writing a real value into a component of an 'IRAS' extension can therefore be performed with a single call to NDF_XPT0R, as follows:

```
CALL NDF_XPT0R( ANGLE, INDF, 'IRAS', 'ANGLE', STATUS )
```

As with the NDF_XGT0x routines (see §11.4), the NDF_XPT0x routines will also accept a compound HDS component name, possibly including array subscripts, as a third argument. This allows the creation of components which are nested more deeply inside the extension, but remember that all HDS structures lying above the new component must already exist. When writing to an array element, the array itself must also previously have been created.

11.6 Accessing Array Information in Extensions

It is relatively common to store quite large amounts of information in NDF extensions and this may include arrays whose size and shape matches that of the main NDF *data* array itself. In this case, the close association between the NDF_ routines and the ARY_ library (SUN/11) provides a number of convenient features for accessing this information.

For example, suppose that an astronomical instrument produces data in NDF format and also produces a measure of (say) temperature for each pixel of the NDF. This temperature information might conveniently be stored as an array within an NDF extension so that it is available during data reduction. The ARRAY data structure, which is handled by the ARY_ routines (a primitive HDS array is an example), would be convenient for this purpose since it also allows arbitrary pixel-index bounds which may be chosen to match those of the associated NDF.

To access the temperature information, we could use the techniques discussed above to locate the array within its extension and then to import it into the ARY_ system. This system provides analogous facilities to the NDF_ routines except that it applies to ARRAY data structures. In particular, the ARY_MAP routine may be used to obtain mapped access to an array, so that the process of accessing the temperature data might be as follows:

```
*  Obtain a locator to the extension.
      CALL NDF_XLOC( INDF, 'INSTR_EXTN', 'READ', LOC, STATUS )

*  Find the array within the extension.
      CALL ARY_FIND( LOC, 'TEMPERATURE', IARY, STATUS )

*  Map the array for access.
      CALL ARY_MAP( IARY, '_REAL', 'READ', PNTR, EL, STATUS )

      <access the mapped array values>

*  Clean up.
      CALL ARY_ANNUL( IARY, STATUS )
      CALL DAT_ANNUL( LOC, STATUS )
```

Here, the integer value IARY is an ARY_ system identifier, analogous to an NDF identifier.

A more convenient method of achieving the same result would be to call the routine NDF_XIARY, which combines the process of locating the extension and finding the array into a single call, as follows:

```
      CALL NDF_XIARY( INDF, 'INSTR_EXTN', 'TEMPERATURE', 'READ', IARY, STATUS )
      CALL ARY_MAP( IARY, '_REAL', 'READ', PNTR, EL, STATUS )

      <access the mapped array values>

      CALL ARY_ANNUL( IARY, STATUS )
```

An additional advantage of this latter method is that NDF_XIARY will check whether the NDF identifier refers to an NDF section and will, if necessary, select a matching section from the temperature array (for a discussion of NDF sections, see §15). NDF_XIARY is also able to accept a compound HDS component name as its third argument, and can therefore access arrays nested more deeply within an extension.

11.7 Deleting Extensions

Specific extensions may be deleted from an NDF using the routine NDF_XDEL. For instance:

```
CALL NDF_XDEL( INDF, 'IRAS', STATUS )
```

would delete the 'IRAS' extension, if present, together with its contents. No error would result if the extension did not exist.

11.8 Enumerating an NDF's Extensions

The routine NDF_STATE may be used to determine whether an NDF contains any extensions by specifying a component name of 'Extension'. Thus:

```
LOGICAL STATE

...

CALL NDF_STATE( INDF, 'Extension', STATE, STATUS )
```

will return a .TRUE. value via the STATE argument if one or more extensions are present in the NDF. The actual number of extensions present can be determined using the routine NDF_XNUMB. For instance:

```
INTEGER NEXTN

...

CALL NDF_XNUMB( INDF, NEXTN, STATUS )
```

will return the number of extensions via the NEXTN argument. The names of any extensions present can also be obtained, in this case using the routine NDF_XNAME. For instance:

```
CHARACTER * ( NDF__SZXNM ) XNAME
INTEGER N

...

CALL NDF_XNAME( INDF, N, XNAME, STATUS )
```

will return the name of the N'th extension in upper-case via the XNAME argument. A blank name will be returned if no such extension exists. Note the use of the NDF__SZXNM symbolic constant (defined in the include file NDF_PAR) to declare the size of the character string which is to receive the extension name.

As an example, the following loop will list the names of all the extensions in an NDF:

```
      CALL NDF_XNUMB( INDF, NEXTN, STATUS )
      DO 1 I = 1, NEXTN
         CALL NDF_XNAME( INDF, I, XNAME, STATUS )
         CALL MSG_SETC( 'NAME', XNAME )
         CALL MSG_OUT( 'NAME', '^NAME', STATUS )
 1    CONTINUE
```

12 ARRAY COMPONENT STORAGE FORM AND COMPRESSION

12.1 General

An NDF data structure allows for the possibility of storing the values of its array components in a variety of different ways within the underlying data system HDS. The reasons for this are various, but have to do with maintaining compatibility with previous data formats and optimising disk space or access time for certain kinds of information. The options are described in SGP/38, where they correspond with the various *variants* of the ARRAY structure, which is one of the building-blocks from which an NDF is constructed.

In the present document, the terminology has been changed slightly. In particular, the term *storage form* is used in preference to *variant* to avoid possible confusion with *variance*, although the meaning is unchanged. Also, note that the term *storage form* incorporates what is often referred to as *compression* - different compression algorithms correspond to different storage forms.

12.2 Obtaining the Storage Form

The storage form of an NDF array component may be determined using the routine NDF_FORM. For instance:

```
INCLUDE 'NDF_PAR'
CHARACTER * ( NDF__SZFRM ) FORM

...

CALL NDF_FORM( INDF, 'Data', FORM, STATUS )
```

will return the storage form of an NDF's *data* component as an upper-case character string via the FORM argument. Note how the symbolic constant NDF__SZFRM (defined in the include file NDF_PAR) should be used to declare the size of the character variable which is to receive the returned storage form information.

The storage form is established when an NDF is first created (see §13.2). At present there is no way of explicitly changing it, but in some circumstances it may be changed implicitly (see §12.6). At present, only four storage forms are supported, so only the values 'SIMPLE', 'SCALED', 'DELTA' and 'PRIMITIVE' can be returned by NDF_FORM. These are described below.

12.3 Simple Storage Form

In this form of storage, the values of an NDF's array component are stored in their simplest possible form, *i.e.* as a sequence of pixels in an N-dimensional array with (optionally) a similar imaginary component. This, together with other ancillary information is held in a single ARRAY structure within HDS.

There are no special restrictions on the use of *simple* arrays and most applications will not need to be aware of the use of this storage form.

12.4 Scaled Storage Form

In this form of storage, the values stored internally within an NDF's array component are a scaled form of the external values of interest to application code. Specifically, the internal scaled values are related to the external unscaled values via:

```
(unscaled value) = ZERO + (scaled value) * SCALE
```

where ZERO and SCALE are two constant values stored with the array. In all other respects, a scaled array is exactly like a simple array.

This storage form is commonly used as a means of compressing the data into a smaller disk file by selecting a data type for the scaled values that uses fewer bytes per value than the unscaled data values (for instance, scaling four byte _REALs into two byte integer _WORDS). Note, information is lost in this process as the original unscaled values cannot be recreated exactly from the scaled value.

Support for scaled arrays is currently limited, since it is anticipated that they will only be of interest as an archive format. The following details should be noted:

(1) Scaled arrays are "read-only". An error will be reported if an attempt is made to map a scaled array for WRITE or UPDATE access. When mapped for READ access, the pointer returned by NDF_MAP provides access to the *unscaled* data values - that is, the mapped values are the result of applying the scale and zero terms to the stored (scaled) values.

    Currently, the scaled (i.e. stored) data values cannot be accessed directly. If you want to change the array values in a scaled array, first take a copy of the NDF and then modify the array values in the copy[11].

(2) The result of copying a scaled array (for instance, using NDF_PROP, *etc.*) will be an equivalent simple array.

(3) Scaled arrays cannot be created directly. To create an NDF with scaled arrays, first create an NDF with simple arrays, and then copy it using NDF_ZSCAL. The output NDF created by NDF_ZSCAL is a copy of the input NDF, but stored with scaled storage form[12].

(4) The NDF_GTSZx routine can be used to determine the scale and zero values of an existing scaled array.

(5) Scaled arrays cannot have complex data types. An error will be reported if an attempt is made to to import an HDS structure describing a complex scaled array, or to assign scale and zero values to an array with complex data values.

(6) When applied to a scaled array, the NDF_TYPE and NDF_FTYPE routines return information about the data type of the unscaled data values. In practice, this means that they return the data type of the SCALE and ZERO constants, rather than the data type of the array holding the stored (scaled) data values. To get the data type of the stored (scaled) values, use NDF_SCTYP.

[11] Copying a scaled array produces an equivalent simple array.

[12] Alternatively, an existing simple NDF can be converted to a scaled array by assigning scale and zero values to it using NDF_PTSZ<T> - a typical program could create a simple array, map it for write access, store the scaled data values in the mapped simple array, unmap the array, and then associate scale and zero values with the array, thus converting it to a scaled array.

12.5 Delta compressed Storage Form

In this form of storage, the values stored internally within an NDF's array component are a compressed form of the external values of interest to application code. Delta form provides a lossless compression scheme designed for arrays of integers in which there is at least one pixel axis along which the array value changes only slowly. It uses two methods to achieve compression:

- Differences between adjacent data values are stored, rather than the full data values themselves. For many forms of astronomical data, the differences between adjacent data values have a much smaller range than the data values themselves. This means that they can be represented in fewer bits. For instance, if the data values are _INTEGER, then the differences between adjacent values may fit into the range of a _WORD (-32767 to +32767) or even a _BYTE (-127 to +127). This use of a shorter data type usually provides the majority of the compression. However, it is not necessary for all differences to be small - if the difference between two adjacent data values is too large for the compressed data type, the second of the two data values will be stored explicitly using the full data type of the original uncompressed data. Obviously, the more values that need to be stored in full in this way, the lower will be the compression.

  In the above description, the term "adjacent" means "adjacent along a specified pixel axis". The pixel axis along which differences are taken is referred to as the "compression axis". It may be specified explicitly by the calling application when NDF_ZDELT is called, or it may be left unspecified in which case NDF_ZDELT will choose the axis that gives the best compression.

- If the uncompressed array contains runs of more than three identical values along the compression axis, then the run of identical values is replaced by a single value (stored in full, not as a difference) and a repetition count.

Support for delta arrays is currently limited, since it is anticipated that they will only be of interest as an archive format. The following details should be noted:

(1) Delta arrays are "read-only". An error will be reported if an attempt is made to map a delta array for WRITE or UPDATE access. When mapped for READ access, the pointer returned by NDF_MAP provides access to the original *uncompressed* data values.

(2) The result of copying a delta array (for instance, using NDF_PROP, *etc.*) will be an equivalent simple array.

(3) Delta arrays cannot be created directly. To create an NDF with delta compressed arrays, first create an NDF with simple arrays, and then copy it using NDF_ZDELT. The output NDF created by NDF_ZDELT is a copy of the input NDF, but stored with delta storage form.

(4) Delta form can only be used to store integer data values, but NDFs with floating point data values may be compressed indirectly, by first storing the floating point values in a scaled NDF, and then using NDF_ZDELT to create a delta compressed copy of the scaled NDF. Note, the scaled NDF must use an integer data type to store the internal (i.e. scaled) values. The use of the scaled NDF means that the compression is not lossless, since some information will have been lost in scaling the floating point values into integers.

(5) The NDF_GTDLT routine will return details of the compression applied to a delta compressed NDF array component.

(6) Delta arrays cannot have complex data types. An error will be reported if an attempt is made to to import an HDS structure describing a complex delta array.

(7) When applied to a delta array, the NDF_TYPE and NDF_FTYPE routines return information about the data type of the original uncompressed data values.

12.6 Primitive Storage Form

This storage form is provided primarily to maintain compatibility with previous data formats. In this case, the array's values are held as a sequence of pixels in an N-dimensional array, but in a *primitive* HDS object. This means that no ancillary information can be associated with such a component and this imposes a number of restrictions on the properties of such arrays:

- The lower pixel-index bounds of all dimensions of an NDF in which primitive arrays are used must be equal to 1 (strictly, this only applies to *base* NDFs, since any NDF *sections* derived from them may still have arbitrary lower bounds – see §15.2).
- Primitive array components cannot hold complex values.
- The *bad*-pixel flag for primitive arrays is always regarded as .TRUE. in the case of an NDF's *data* and *variance* components.

In most situations, these restrictions are unimportant and *primitive* storage form may be used to maintain compatibility with existing datasets and software. In the longer term, it is expected that a gradual transition will take place, replacing *primitive* arrays by equivalent *simple* arrays and this latter approach should be taken by all new software. However, there is usually little harm in creating NDFs with *primitive* array components, because any change made to the NDF which would violate one of the restrictions above will cause any affected *primitive* array component to implicitly change its storage form to become *simple*. This is a straightforward change which costs little, and a subsequent call to NDF_FORM will show if this has occurred. Only one possible complication may arise: if the array is mapped for access when its storage form is implicitly changed, then an error will result. This is unlikely to be a problem in practice.

Warning – possible pitfall: A case which occasionally causes problems can arise if a *primitive* NDF is created (*e.g.* by calling NDF_CREP – see §13.4) and an array component is then mapped for access using an access mode such as 'WRITE/ZERO'. This access mode will cause the component's *bad*-pixel flag to be set to .FALSE. (see §9.7). When the component is unmapped, this, in turn, will cause its storage form to be implicitly converted to *simple*.

This behaviour is correct, but it is not always what is expected, or wanted. It can be avoided by setting the *bad*-pixel flag value back to .TRUE. (see §9.6) before unmapping the component concerned, or by performing the initialisation to zero explicitly rather than via an initialisation option on the mapping mode.

13 ACCESSING NDFS FOR OUTPUT

13.1 Using Existing NDFs

An NDF data structure which is to be used for output from an application can be obtained in a number of ways. Most simply, an existing NDF may be acquired by using the NDF_ASSOC routine (§3.1) and specifying 'UPDATE' or 'WRITE' access (in both the call to the routine and the associated interface file if you are using ADAM). New values, or new components, may then be added to the NDF, as in the following simple example which obtains a new value for an NDF's *title* component:

```
SUBROUTINE TITLE( STATUS )
INTEGER STATUS, INDF

IF ( STATUS .NE. SAI__OK ) RETURN
CALL NDF_ASSOC( 'OUT', 'UPDATE', INDF, STATUS )
CALL NDF_CINP( 'TITLE', INDF, 'Title', STATUS )
CALL NDF_ANNUL( INDF, STATUS )
END
```

The same principle would also be used to write a new NDF *data* component by mapping it for 'WRITE' access and assigning new values to its pixels, as described in §8.4.

Note that 'UPDATE' access has been used here in the call to NDF_ASSOC because we want other NDF components to retain their previous values. If 'WRITE' access had been specified, then the NDF's components would have been automatically reset to an undefined state, as the NDF_ system interprets this access mode as a request to *re*-write the data structure.

13.2 Creating New NDFs via Parameters

An alternative method of obtaining an NDF for output is to create an entirely new one. The NDF_CREAT routine will perform this task. For instance:

```
INTEGER NDIM, LBND( NDIM ), UBND( NDIM )

...

CALL NDF_CREAT( 'OUT', '_REAL', NDIM, LBND, UBND, INDF, STATUS )
```

will create a new NDF and associate it with the parameter 'OUT'. If you are using ADAM, a similar interface file entry to that in §3.1 would be required, except that 'WRITE' access would be specified in this case.

This example will create a *simple, real* NDF; *i.e.* one whose *data* and *variance* components will be stored as '_REAL' arrays, and whose array components will have a storage form of 'SIMPLE' (see §12.3). Its dimensionality and pixel-index bounds are specified by the NDIM, LBND and UBND arguments. Initially, all its components will be in an undefined state.

13.3 Conditional NDF Creation

It is sometimes necessary to determine whether an NDF exists before deciding to create a new one. The routine NDF_EXIST is provided to allow this by associating an existing NDF with a parameter and returning an NDF identifier for it, if it exists. If the NDF does not exist, then no error results, but the routine returns with a "null" identifier value of NDF__NOID (defined in the include file NDF_PAR). In effect, this routine behaves identically to NDF_ASSOC, except that if the NDF does not exist, control is returned to the calling routine rather than re-prompting the user to supply a new name. The following illustrates how NDF_EXIST might be used:

```
INCLUDE 'NDF_PAR'

...

CALL NDF_EXIST( 'OUT', 'UPDATE', INDF, STATUS )
IF ( INDF .EQ. NDF__NOID ) THEN
   CALL NDF_CREAT( 'OUT', '_INTEGER', NDIM, LBND, UBND, INDF, STATUS )
END IF
```

Here, an existing NDF is accessed if it exists, otherwise a new structure is created and used instead.

13.4 Creating Primitive NDFs

The routine NDF_CREP is provided for creating *primitive* NDFs; *i.e.* NDFs whose array components will have a storage form of 'PRIMITIVE'. NDF_CREAT should normally be used to create a new NDF, but use of NDF_CREP may sometimes be necessary in order to maintain compatibility with existing software (see §12.6). Because of the restrictions inherent in the primitive form of array storage, the lower pixel-index bounds of primitive NDFs must be set to 1 in all dimensions, so NDF_CREP lacks the LBND argument of NDF_CREAT, the appropriate lower bounds being assumed:

```
CALL NDF_CREP( 'OUT', '_REAL', NDIM, UBND, INDF, STATUS )
```

Its use is otherwise identical to NDF_CREAT.

14 COMPONENT PROPAGATION

14.1 General

New output NDFs may also be generated by a process termed *propagation*, in which a new structure is created based on an existing *template* NDF. This is the most common method of creating an NDF to contain output from a processing algorithm, and is typically used whenever an application draws input from one or more NDFs and produces a new output NDF as a result.

As far as the user of such applications is concerned, the output dataset would typically be based upon one of the input datasets; *i.e.* it might inherit its shape and component type, storage form

and possibly values from an input dataset.[13] Of course, the output data structure would also incorporate whatever changes the processing algorithm is designed to perform.

Seen from within such an application, the purpose of propagation is to create a "skeleton" output NDF based on an input structure, but containing "blank" (*i.e.* undefined) components into which calculated results can be inserted. Usually, there will also be "non-blank" (*i.e.* defined) components in the newly-created NDF, which derive their values directly, without change, from one of the input datasets. Such components are said to have been *propagated*.

The way in which components (and extensions) are selected for propagation is central to the philosophy of the NDF and it is important to understand the principles if you are to write applications which process NDFs consistently. There are two sets of *propagation rules* which apply separately to standard NDF components and to extensions. The distinction between them is explained in the following two sections, after which the way in which these ideas are implemented in practice using NDF_ routines is described.

14.2 Propagation Rules for Standard NDF Components

Because the meaning and interpretation of the standard NDF components (*data*, *variance*, *quality*, *etc.*) is well-defined, it is always possible to decide into which of three categories each of these components falls when writing an application. This then dictates the action which should be taken, as follows:

(1) **Process it.** *Any component which an application is capable of processing must be processed in such a way that its validity is maintained.*

> For instance, if an operation is to be performed on the *data* component, then an appropriate operation must usually also be performed on the *variance* component (if defined) so that it continues to represent the variance of the data in the output NDF. If the application cannot perform the necessary operation, then any component which would become invalid as a result falls into category 3 below (and is simply ignored).

(2) **Propagate it.** *Any components which will not be rendered invalid by the processing can be propagated without change.*

> For instance, the special case of adding a constant to the *data* component would not render the *variance* component invalid. It may therefore simply be propagated (*i.e.* copied) to the output NDF unchanged. Most applications which perform pixel-to-pixel processing and do not change the shape of an NDF can also propagate the *axis* and *quality* components in the same way.

(3) **Ignore it.** Any remaining components whose subsequent validity is in any doubt must be ignored, never propagated.

> It may not be possible to ensure that some NDF components will retain their validity after processing. This may simply be because an application (or perhaps an entire software package) chooses not to support certain NDF components; this is quite acceptable behaviour. Alternatively, the meaning of certain components may be destroyed by

[13] Where there is more than one input NDF, one of them should be designated the *primary* input dataset and be used as the template for the output dataset. By convention, this should be the first input NDF acquired by the application and the first to be described in documentation.

certain types of processing; for instance the validity of the *quality* component cannot possibly survive a Fourier transform operation applied to the *data* component no matter how sophisticated the software. In either case, the affected component(s) must be ignored and not propagated. This means that they will be lost from the output NDF.

The purpose of these rules is to ensure that the validity of all the standard NDF components is retained throughout all stages of processing, and that all defined components in an NDF always have valid values. If an application cannot guarantee this for any component, then it must ignore that component so that it remains undefined in the output data structure.

14.3 Propagation Rules for Extensions

The propagation of extensions is necessarily different from the propagation of standard NDF components, because only certain pieces of software may recognise any particular extension. There is therefore no way in which other applications can judge whether the processing they are performing will render the information in the extension invalid. This is an unavoidable consequence of extensibility.

Rather than taking a pessimistic view and automatically deleting all unrecognised extensions, propagation of extensions proceeds along more optimistic lines, as follows:

(1) **Process it.** *If an application recognises an extension, and can process it, then it should ensure that its validity is maintained.*

   For instance, an application in a software package which recognised an 'IRAS' extension should always check for the existence of such an extension and ensure that its contents were not invalidated by the processing it performs, making changes to the extension to achieve this if necessary.

(2) **Suppress it.** *If an application recognises an extension but cannot process it, then propagation should be suppressed.*

   For instance, an application may recognise an extension but be unable to ensure its continued validity after processing, either due to a limited implementation or more fundamental causes. In either event, the application should suppress propagation of that extension so that it is lost from the output data structure.

(3) **Propagate it.** *Any extension which is not recognised will be propagated by default.*

   There may be any number of extensions present in an NDF which a particular application does not recognise. These should be ignored, and the default action of the NDF_ system will then be to propagate (*i.e.* copy) them to the output NDF.

The purpose of these rules is to ensure that extension information is always retained unless it is certain that it will no longer be valid. This contrasts with the rules for processing standard NDF components (§14.2) which are retained only if it is certain that they will still be valid.

Of course, the rules for propagating extensions carry the risk that an application which does not recognise an extension will inadvertently render it invalid. However, if all applications within a software package recognise the same extension(s), then this can only happen if software from several packages is intermixed. It then becomes the user's responsibility to check the validity of information held in extensions.

14.4 Creating New NDFs by Propagation

Fortunately, the rules above are far easier to apply in practice than they might appear, and typically amount simply to deciding which NDF components (or extensions) will not need any processing performed on them. These components are propagated and any that remain (and which the application chooses to support) are then processed. All others are ignored.

Propagation is performed by the routine NDF_PROP, which creates a new NDF structure via a parameter, based on a template NDF which already exists. At the same time, it will selectively copy components and extensions present in the template structure and use them to initialise the corresponding components (and extensions) in the new NDF. For example:

```
CALL NDF_PROP( INDF1, ' ', 'OUT', INDF2, STATUS )
```

will create a new output NDF, associate it with the parameter 'OUT' and return an identifier for it via the INDF2 argument. The new NDF is based on the template structure with identifier INDF1, and inherits its shape and the type and storage form of its components. Subsequent changes may be applied to any of these inherited attributes if required, *e.g.* by calling NDF_STYPE to change the numeric type of any of the new NDF's array components (see §7.4).

14.5 Default Propagation of Components and Extensions

The second (CLIST) argument to NDF_PROP specifies a list of components and extensions which are to have their values propagated (*i.e.* copied) to initialise the new NDF. The default (blank) value specified above causes all extensions, together with the *title*, *label* and *history* components to be propagated, if present. These three standard components are considered "safe", in that they are likely to retain their validity through most common types of processing which takes place on NDFs.

Note, however, that no *axis*, *data*, *variance*, *quality* or *units* information will be propagated by default. This allows an application to explicitly process these components to generate new versions for the output structure if possible, or simply to ignore them and have them omitted from the output NDF if the necessary processing cannot be performed.

14.6 Forcing Component Propagation

If you are certain that a component will not be rendered invalid by the processing which an application performs (for instance the addition of a constant to the *data* component would leave the *axis*, *quality* and *variance* components valid), then propagation of these components can be specified by listing them in the CLIST argument to NDF_PROP, thus:

```
CALL NDF_PROP( INDF1, 'Axis,Qual,Var', 'OUT', INDF2, STATUS )
```

These components would then be copied to the output NDF and would not need to be explicitly considered again by the application.

14.7 Inhibiting Component Propagation

Conversely, if any of the standard NDF components which are propagated by default would be rendered invalid by the processing an application performs, then propagation may be inhibited by specifying the component name with the prefix 'NO' in the CLIST argument to NDF_PROP. For instance:

```
CALL NDF_PROP( INDF1, 'Quality,NoLab', 'OUT', INDF2, STATUS )
```

would force propagation of the *quality* component, but inhibit the default propagation of the *label* component.

14.8 Controlling Propagation of Extensions

Propagation of specific extensions may also be inhibited by specifying 'NOEXTENSION()' in the CLIST argument to NDF_PROP and listing the extensions to be omitted between the parentheses. For instance:

```
CALL NDF_PROP( INDF1, 'Axis,Noext(IRAS,ASTERIX)', 'OUT', INDF2, STATUS )
```

would force propagation of the *axis* components, but inhibit the propagation of any 'IRAS' or 'ASTERIX' extension which may be present. Note that 'NOEXTENSION' may be abbreviated but extension names must appear in full, although mixed case is permitted.

This mechanism can be useful if an extension may contain a large amount of information but is not required in the output data structure. The alternative (of propagating the extension, then deleting it) is less efficient and may lead to an unnecessarily large output file.

An asterisk may be used to indicate "all extensions". Thus

```
CALL NDF_PROP( INDF1, 'Axis,Noext(*)', 'OUT', INDF2, STATUS )
```

would inhibit the propagation of all extensions. Likewise,

```
CALL NDF_PROP( INDF1, 'Axis,Noext(*),Ext(IRAS)', 'OUT', INDF2, STATUS )
```

would propagate just the IRAS extension.

15 NDF SECTIONS

15.1 The Need for NDF Sections

An important facility provided by the NDF_ system is the ability to select a region from an NDF which differs in shape from the original NDF, and to process it as if it were a complete NDF itself. Such regions are termed *NDF sections*. As a simple example, consider a routine which plots a contour map of an NDF's *data* component. By accessing an appropriate NDF section, the

same routine could also be used to contour only a subset of the data without having to make any change to the routine itself.

This ability to concentrate on a subset of an NDF can clearly improve efficiency in many circumstances, but NDF sections are also capable of referring to *super-sets* of NDFs; *i.e.* they may extend beyond the bounds of the NDF from which they are derived. This gives them a number of further uses through their ability to match the shapes of NDFs of otherwise unequal extent by effectively trimming or padding them with *bad* pixels. Their use for this type of operation is discussed more fully in §17.3.

15.2 Creating NDF Sections

An NDF section may be created using the routine NDF_SECT. For instance:

```
INTEGER NDIM, LBND( NDIM ), UBND( NDIM )

...

CALL NDF_SECT( INDF1, NDIM, LBND, UBND, INDF2, STATUS )
```

will create an NDF section starting from an NDF with identifier INDF1, and will return a new identifier INDF2 which refers to the section. The set of pixels in the original NDF to which the new section should refer is determined by the arguments NDIM, LBND and UBND.

These arguments not only specify the set of original pixels to which the new section should refer, but also directly determine the *shape* (*i.e.* the pixel-index bounds and the dimensionality) of the new section. For instance, a call to NDF_BOUND using the new identifier INDF2 would return the same values as had originally been specified in the call to NDF_SECT.[14]

Note, the tuning parameter SECMAX places a limit on the largest possible NDF section that can be created (see §23.3).

15.3 The Distinction between Base NDFs and Sections

An NDF identifier which refers to an entire NDF dataset (not just a section of it) is said to refer to a *base* NDF. A base NDF represents a data structure whose shape and other attributes are uniquely defined. Any changes made to the attributes of a base NDF are reflected by actual changes to the contents of the data file in which the NDF is stored. Any such changes are also immediately apparent through any other base NDF identifiers which refer to the same structure (multiple identifiers for the same structure can be generated by means of the routine NDF_CLONE for instance).

In contrast, an NDF section simply represents a "window" into a base NDF. Any number of identifiers may refer to sections derived from the same base NDF, but each represents a separate window. As a consequence, the *shapes* of all NDF sections are completely independent and may be altered, if required, without affecting each other. One consequence of this is that the NDF_CLONE routine, when applied to an NDF section, has the effect of producing a duplicate NDF *section* rather than simply a duplicate NDF *identifier*.

[14] Users familiar with HDS should note the distinction between the HDS concept of *slicing*, in which the pixel indices of an array slice always start at (1,1. . .), and the use of *sections* from NDFs, where the pixel indices of the original NDF are preserved in any derived section.

It is possible to discover if an identifier refers to a base NDF or to an NDF section using the routine NDF_ISBAS. For instance:

```
CALL NDF_ISBAS( INDF, ISBAS, STATUS )
```

will return a logical value of .TRUE. via the ISBAS argument if the identifier INDF refers to a base NDF. It is also possible to obtain an identifier for the base NDF to which an NDF section refers by using the routine NDF_BASE, thus:

```
CALL NDF_BASE( INDF1, INDF2, STATUS )
```

However, this routine should be used sparingly because it tends to subvert the program modularity which the use of NDF sections allows, and permits access to regions of the base NDF which a well-structured application perhaps ought not to be touching.

15.4 Referring to Subsets and Super-sets

In describing how to create an NDF section in §15.2, there was an implicit assumption that the pixel-index bounds of the section lay within the bounds of the NDF from which it was derived, and that the section's dimensionality also matched that of the original NDF. In fact, neither of these restrictions need apply.

First consider the case where the dimensionality of the initial NDF and the derived section are the same, but the new pixel-index bounds extend outside those of the original NDF. This would be the case if an NDF section with shape:

(1:256, 1:512)

were to be created from an original NDF with shape:

(0:511, 10:300)

In this case the section refers to a *subset* of the pixels along the first dimension but a *super-set* of the pixels along the second dimension. As a consequence, there are some pixels in the new section which do not exist in the original NDF.

When values are read from an array component of such a section, the NDF_ system will respond by *padding* the original NDF with *bad* pixels; *i.e.* by assigning the appropriate *bad*-pixel value to all the "new" pixels which did not exist in the original NDF. The value of the *bad*-pixel flag returned for the new section by the routine NDF_BAD would reflect the presence of these *bad* pixels.

15.5 The Transfer Window

The set of pixels which lie within the bounds of a base NDF and also within the bounds of a section derived from it is termed the *transfer window* for that section. This transfer window comes into play when new values are assigned to the section's pixels. Although new values may be assigned to any of these pixels, only those lying within the transfer window will have their values transferred back to the base NDF to cause a permanent change to the data structure.

The transfer takes place when access to a mapped array component of a section is relinquished (*e.g.* by calling NDF_UNMAP), at which point all pixel values outside the transfer window are discarded.

This process is similar in concept to the "viewport" used in many graphics systems where, typically, a program can plot lines at any point, but only those lying within the viewport will actually appear on the screen (*i.e.* plotting done outside the viewport is "clipped"). Using this analogy, an NDF section would correspond with the plotting space available to a program, the transfer window would correspond with the viewport, and the base NDF would correspond with the plotting screen.

Note that it is quite permissible for an NDF section to be derived from another NDF section without any restriction on their relative pixel-index bounds. In this case, the transfer windows of both sections will be combined, so that no new section can access a larger region of the associated base NDF than the section from which it is derived. In extreme cases this could result in the transfer window for a section becoming non-existent, in which case the section will no longer have any contact with its base NDF, although it will still be a valid section.

15.6 Changing Dimensionality

Now consider the case where the number of dimensions specified for a new NDF section differs from the dimensionality of the NDF from which it is derived. The NDF_ system will handle this in the same way that all dimensionality mis-matches are handled; *i.e.* by padding the pixel-index bounds with 1's as necessary.

For example, suppose a 1-dimensional section with shape:

(3:10)

were to be derived from a 2-dimensional NDF with shape:

(1:20, 1:20)

In this case, the 1-dimensional shape would first be padded with 1's to become:

(3:10, 1:1)

which identifies the pixels to which the new section should refer. The additional 1's will then be discarded before the section is created so that a 1-dimensional section results. A similar process would take place if the relative dimensionalities were reversed, but it would then be the original NDF's pixel-index bounds which were padded with 1's in order to identify the pixels to which the section should refer.

There are no restrictions on the creation of sections of any dimensionality up to the maximum of 7 supported by the NDF_ routines. Changes of dimensionality may also be freely combined with the selection of super-sets (see §15.4).

15.7 Restrictions on the Use of Sections

In general, an identifier for an NDF section can be passed without error to any routine which will accept an equivalent identifier referring to a base NDF. In certain cases, however, the behaviour of the routine may differ slightly when an NDF section is supplied in order to adhere to two guiding principles:

(1) It should be straightforward to write applications which process NDFs without having to know whether the NDFs concerned are base NDFs or NDF sections. Having written such an application, it should be able to process either without modification.

(2) Applications which access NDF sections should not generally cause changes to the values of NDF pixels which lie outside the pixel-index bounds (or more accurately the transfer window) of the sections they are processing.

The set of operations affected by these principles is rather small because most NDF components (*e.g. label*, *units*, *title*, *history* and *extensions*) are regarded as *global* and are equally accessible via identifiers referring to NDF sections and base NDFs. It is mainly operations on array components which behave differently when applied to NDF sections, and notably those which affect the attributes of these components. For instance, the numeric type of an NDF array component cannot be changed using NDF_STYPE (§7.4) via a section identifier; instead, this routine will simply return without action. Neither may an NDF be deleted (§20.11) via a section identifier.

These, and other differences, are noted in the appropriate routine descriptions in Appendix D and at other relevant points in this document.

15.8 Restrictions on Mapped Access to Sections

The restrictions described in §8.10 concerning multiple mapped access to NDF array components also apply to NDF sections if there is a possibility of an access conflict occurring. Thus, if two NDF sections refer to the same base NDF, and the regions to which they refer (more precisely their transfer windows) intersect, then only one of these sections may be mapped at any time for write or update access to the same array component.

If necessary, an application can determine if such a conflict may occur by using the routine NDF_SAME. For instance:

```
LOGICAL SAME, ISECT

...

CALL NDF_SAME( INDF1, INDF2, SAME, ISECT, STATUS )
```

will return a .TRUE. value via the SAME argument if the two NDF identifiers supplied refer to the same base NDF, and will also return a .TRUE. result via the ISECT argument if their transfer windows intersect.

15.9 More Advanced Use: Partitioning

In times gone by, when computers had little memory, it was common for image-processing applications to read their data one line at a time, and to operate on that line before passing on to the next. Nowadays, there is little need for this approach and the programming complications it causes, because reasonably-sized images can be accommodated entirely in memory.

Nevertheless, limitations on the available memory can still be important in some situations. For instance, when processing extremely large files, it is still possible to find that the memory available (or the memory quota allocated by your system manager) is insufficient. Even when it appears possible to accommodate a large array in memory, it is wise to remember that with a *virtual memory* operating system the information may not actually reside in the physical memory of the computer, and this may result in substantial inefficiencies.

This sort of consideration can be ignored for most types of work, but steps must sometimes be taken to reduce the amount of memory required when accessing large NDFs. This will often result in improvements in efficiency even if actual memory limitations are unimportant, because a reduction in memory requirements in general tends to make more memory available for other system activities, which therefore run more efficiently.

To allow memory usage to be limited, the NDF_ system provides facilities for partitioning NDFs so that they may be processed in pieces. Two partitioning methods are available, termed *chunking* and *blocking*. *Chunking* is appropriate when the NDF can be regarded simply as a 1-dimensional sequence of values (*i.e.* when the actual shape is unimportant) while *blocking* is used if the shape and positional relationship between the pixels is significant. Both of these techniques work by dividing an NDF into sections. They are described in turn below.

Note that these techniques may also have applications in parallel processing, where it is often necessary to partition a large array and then to pass the resulting pieces to separate processors.

15.10 Chunking

The technique of *chunking* is best introduced by an example. Consider a large 1-dimensional NDF (a spectrum, perhaps) which is to be processed in pieces in order to limit memory usage, and suppose that the maximum size of one of these pieces is to be 10000 pixels. The spectrum can be divided into pieces for processing by creating a section to refer to each piece. Each of these sections should follow on from the previous one, and each should contain 10000 pixels (except the last one, which may be smaller if the total number of pixels is not an exact multiple of 10000). Each of these sections is termed a *chunk*.

If we want to process each of these *chunks* in turn, we need to know how many there are, and to have a method of creating the appropriate sections. The NDF_ system provides these facilities through the routines NDF_NCHNK (which determines the number of *chunks* available) and NDF_CHUNK (which creates sections referring to successive *chunks*). The following illustrates how these routines might be used:

```
      INTEGER ICHUNK, ICH, MXPIX, NCHUNK
      PARAMETER ( MXPIX = 10000 )

      ...
```

```
*  Determine the number of chunks available.
      CALL NDF_NCHNK( INDF, MXPIX, NCHUNK, STATUS )

*  Loop through the chunks, creating a section to refer to each one.
      DO 1 ICHUNK = 1, NCHUNK
         CALL NDF_CHUNK( INDF, MXPIX, ICHUNK, ICH, STATUS )

         <access the resulting section/chunk>

*  Annul the section identifier.
         CALL NDF_ANNUL( ICH, STATUS )
 1    CONTINUE
```

Here, MXPIX is set equal to the maximum number of pixels which a *chunk* is to contain, and NDF_NCHNK is then called to determine how many such *chunks* are available in the NDF (the result is returned via the NCHUNK argument). The routine NDF_CHUNK is then called repeatedly to create a sequence of NDF sections which refer to each *chunk* in turn. These *chunks* are specified by the *chunk index* ICHUNK, which varies from 1 to NCHUNK. The NDF identifier for each section created by NDF_CHUNK is annulled when it is no longer required.

In the 1-dimensional case (as here), this process of *chunking* is straightforward, and could have been programmed directly without too much difficulty. However, in the N-dimensional case (with $N > 1$), the number of pixels in each NDF section (*chunk*) must be the product of N separate dimension sizes. Thus, not all sizes of *chunk* are available. In addition, each *chunk* must fit within the bounds of the NDF from which it is derived. As a result, the size and shape of each *chunk* returned by NDF_CHUNK may vary (although the sequence of *chunks* generated is always repeatable if several NDFs with the same shape are processed using the same value of MXPIX).

The purpose of *chunking* is to divide an NDF into pieces, each of which contains *contiguous* pixels (*i.e.* pixels whose storage locations in the NDF follow one after the other) with the *chunks* themselves also following each other contiguously in pixel-storage order.[15] NDF_CHUNK will accomplish this for any shape of NDF, and for any limit on the maximum number of pixels in a *chunk*. Thus, by appropriately defining MXPIX, a limit can be set on memory usage regardless of the total size of NDF being processed.

In fact, by setting MXPIX to certain special values it is possible to partition an NDF into *chunks* of pre-determined shape. For instance, if MXPIX is set equal to the first dimension size, then NDF_CHUNK will step through the NDF one "line" at a time. Similarly, if MXPIX is set to the product of the first two dimension sizes, then NDF_CHUNK will step through each "plane" of a 3-dimensional stack of images. The following example shows how this might be used to apply a smoothing algorithm to each image in a 3-dimensional stack without needing to access the entire NDF at once:

```
*  Obtain the input NDF shape and set MXPIX.
      CALL NDF_DIM( INDF, NDF__MXDIM, DIM, NDIM, STATUS )
      MXPIX = DIM( 1 ) * DIM( 2 )

*  Create the output NDF, using the input NDF as a template.
      CALL NDF_PROP( INDF1, ' ', 'OUT', INDF2, STATUS )
```

[15]The pixels will be contiguously stored whenever a base NDF is partitioned into *chunks*. However, an NDF section may also be partitioned in the same way. In this case, the resulting "*chunked*" pixels will only be truly contiguous if they were stored contiguously in the original NDF section.

```
*  Determine the number of chunks (i.e. image planes) and loop through
*  them.
      CALL NDF_NCHNK( INDF1, MXPIX, NCHUNK, STATUS )
      DO 1 ICHUNK = 1, NCHUNK

*  Start a new NDF context and create sections for the input/output
*  image planes.
         CALL NDF_BEGIN
         CALL NDF_CHUNK( INDF1, MXPIX, ICHUNK, ICH1, STATUS )
         CALL NDF_CHUNK( INDF2, MXPIX, ICHUNK, ICH2, STATUS )

*  Access the data.
         CALL NDF_MAP( ICH1, 'Data', '_REAL', 'READ', PNTR1, EL, STATUS )
         CALL NDF_MAP( ICH2, 'Data', '_REAL', 'WRITE', PNTR2, EL, STATUS )

         <perform the smoothing operation>

*  End the NDF context.
         CALL NDF_END( STATUS )
 1    CONTINUE
```

Note the use of an NDF context to simplify "cleaning up" after processing each *chunk*.

15.11 Blocking

The concept of *blocking* is similar to *chunking*, except that it is appropriate when the relative positions of pixels within an NDF are significant. In one dimension, *chunking* and *blocking* are equivalent, so a 2-dimensional example is required for illustration.

Suppose that a contour map of a very large 2-dimensional image is to be generated, and that the image must be divided into pieces to limit memory usage. *Chunking* would not be appropriate here, because (depending on its shape) a *chunk* might work out to be a single line of the image, and this would result in inefficient contouring. Rather than setting an upper limit on the total size of each piece, what we need to limit here is the size of each dimension. This means that we must "tile" the 2-dimensional image with a series of adjacent rectangular regions, each of which does not exceed a certain size in each dimension, and each of which can be contoured in turn. This form of partitioning is what *blocking* provides, the "tiles" (in N dimensions) being termed *blocks*.

Blocking is supported by two routines analogous to those provided for *chunking*. NDF_NBLOC calculates the number of *blocks* available in an NDF (for given limits on the dimension sizes), while NDF_BLOCK creates the NDF sections which refer to the individual *blocks*, each of which is identified by a *block index*. The following illustrates the principle:

```
      INTEGER IBLOCK, IBL, MXDIM( 2 ), NBLOCK
      DATA MXDIM / 100, 100 /

      ...

*  Determine the number of blocks available.
      CALL NDF_NBLOC( INDF, 2, MXDIM, NBLOCK, STATUS )
```

```
*  Loop through the blocks, creating a section to refer to each one.
      DO 1 IBLOCK = 1, NBLOCK
         CALL NDF_BLOCK( INDF, 2, MXDIM, IBLOCK, IBL, STATUS )

         <contour the resulting section/block>

*  Annul the section identifier.
         CALL NDF_ANNUL( IBL, STATUS )
 1    CONTINUE
```

In this 2-dimensional example, each *block* is constrained so as not to exceed 100 pixels in size in each dimension, these limits being specified in the MXDIM array.

Note that *blocking* generates sections which lie adjacent to one another, rather than being contiguously stored, as is the case with *chunking*. *Blocks* are numbered so that their lower/upper bounds increase in the conventional sense with increasing *block index* (*i.e.* with the first dimension bound increasing most rapidly and the last bound increasing least rapidly). As with *chunking*, the size of each *block* generated by NDF_BLOCK may vary in order to lie within the original NDF (using the "tiling" analogy, there may be some tiles at the edges which must be "cut" to fit), but the sequence of *blocks* generated is always repeatable.

By supplying special values to NDF_BLOCK for the MXDIM value of each dimension, it is also possible to step through an NDF in *blocks* of a pre-determined shape (as with *chunking*). For instance, if the integer array DIM holds the original NDF dimensions, then the assignment:

```
      MXDIM( 1 ) = DIM( 1 )
```

could be used to step through an image in "lines", while the assignment:

```
      MXDIM( 1 ) = DIM( 1 )
      MXDIM( 2 ) = DIM( 2 )
```

would step through a 3-dimensional image stack in "planes". The assignment:

```
      MXDIM( 1 ) = 1
      MXDIM( 2 ) = DIM( 2 )
```

could also be used to step through an image in "columns", but note that this non-sequential mode of access may not be efficient.

16 USING SUBSCRIPTS TO ACCESS NDF SECTIONS

As well as providing the programmer with explicit facilities for creating NDF sections, the NDF_ system also allows both the user and the writer of applications to specify sections (subsets and super-sets) when giving the names of NDF data structures to be processed. To see how this works, consider an application which requests access to an NDF data structure as follows:

```
      CALL NDF_ASSOC( 'IN', 'READ', INDF, STATUS )
```

and suppose this results in a prompt asking for the name of an NDF:

```
IN - Input NDF data structure > name
```

If you were to respond simply with the name of an HDS object (denoted here by "name"), then NDF_ASSOC would return a base NDF identifier for the specified data structure via its INDF argument. However, if a set of subscripts is also supplied, thus:

```
IN - Input NDF data structure > name(3:256,-8:4)
```

then NDF_ASSOC will return an identifier for the specified NDF section instead.

This process may be applied when accessing a pre-existing data structure in any situation where an NDF name alone would suffice (*e.g.* on the command line when invoking an application, or as a default in an interface file, *etc.*). Writers of applications may also use this method of selecting NDF sections when passing the names of datasets to NDF_ routines which access them (see §20.1).

Note, the tuning parameter SECMAX places a limit on the largest possible NDF section that can be created (see §23.3).

16.1 Specifying Lower and Upper Bounds

The subscript expression appended to an NDF name to specify a section may be given in several ways. One possible method, corresponding with the example above, is to give the lower and upper bounds in each dimension, as follows:

```
name( a:b, c:d, ...  )
```

where '`a:b`', '`c:d`' (*etc.*) specify the lower and upper bounds. The bounds specified need not necessarily lie within the actual bounds of the NDF, because *bad* pixels will be supplied in the usual way, if required, to pad out the NDF's array components whenever they are accessed. However, none of the lower bounds should exceed the corresponding upper bound.

Omitting any of the bounds from the subscript expression will cause the appropriate (lower or upper) bound of the NDF to be used instead. If the separating ':' is also omitted, then the lower and upper bounds of the section will both be set to the same value, so that a single pixel will be selected for that dimension. Omitting the bounds entirely for a dimension (but still retaining the comma) will cause the entire extent of that dimension to be used. Thus,

```
image(,64)
```

could be used to specify row 64 of a 2-dimensional image, while:

```
cube( 1, 257:, 100 )
```

would specify column 1, pixels 257 onwards, selected from plane number 100 of a 3-dimensional "data cube". Note that specifying:

```
image(,)
```

is just another way of referring to an entire image, except that it actually generates a *section* containing all the NDF's pixels.

16.2 Specifying Centre and Extent

An alternative form for the subscript expression involves specifying the centre and extent of the region required along each dimension, as follows:

```
name( p~q, r~s, ... )
```

where 'p~q', 'r~s', (*etc.*) specify the centre and extent. Thus,

```
name(100~11,200~5)
```

would refer to an 11 x 5 pixel region of an image centred on pixel (100, 200).

If the value before the delimiting '~' is omitted, it will default to the index of the central pixel in that dimension (rounded downwards if there are an even number of pixels). If the value following the '~' is omitted, it will default to the number of pixels in that dimension. Thus,

```
image( ~100, ~100)
```

could be used to refer to a 100 x 100 pixel region located centrally within an image, while

```
image( 10~, 20~ )
```

would specify a section which is the same size as the original image, but displaced so that it is centred on pixel (10, 20).

16.3 Using WCS Coordinates to Specify Sections

Most of the previous examples use pixel indices to define the required NDF section. However, sections may also be specified within the coordinate system represented by the current Frame in the NDFs WCS FrameSet. Pixel indices should always be supplied as integer values within subscript expressions (that is, they should not include a decimal point). Anything that is non-integer will be interpreted as a WCS value.

In previous versions of the NDF library, non-integer values were interpreted as *axis* coordinates rather than WCS coordinates (for a description of *axis* coordinates, see §18). In order to retain backwards compatibility, the current version of the NDF library will continue to use this convention if the NDF has an explicitly defined *axis* coordinate system. Otherwise, non-integer values will be interpreted as current WCS Frame values.

The AST_UNFORMAT method is used to read each string representing an axis value. Thus the strings should be supplied in a format which AST_UNFORMAT can understand. Some classes of coordinate system (*e.g.* celestial longitude and latitude, or time) can use a colon':' as a separator between sexagisimal field. Since a colon is also used to separate the upper and lower bounds in an NDF subscript expression, steps must be taken to differentiate the two uses of the colon character. This can be done in two ways:

(1) One method is to use alternative sexagisimal field separators. For instance time (and Right Ascension) values can use use "hms" separators in place of colon separators. Thus an RA value of "12:02:1.1" can also be supplied as "12h02m1.1s". Similarly, a Declination value of "-23:45:12.2" can also be supplied as "-23d45m12.2s".

(2) Another method is to use semi-colon ";" instead of colon ":" as the NDF subscript expression separator. This works with all classes of coordinate system, but leaves some degree of ambiguity. For instance, a subscript expression of "12:34" could mean "use pixel indices 12 to 34", or could mean "use the single RA or Dec value 12:34". In cases, where such ambiguity is significant, sexagisimal fields should be identified explicitly using the appropriate "dhms" separator, as described in the previous point.

If an NDF section is specified as "$(L_1 : U_1, L_2 : U_2, ...L_i, U_i)$", the axis values L_i and U_i will be interpreted as pixel index values for pixel axis i if the values are integer. If the values are non-integer, they will be interpreted as values for WCS axis i. The distinction between WCS axis I and pixel axis I is important since pixel axis I does not necessarily correspond to WCS axis I (for instance the WCS axes may be rotated or permuted).

If the section is specified by centre and extent (see §**??**), the extent will be interpreted as a number of pixels if it is an integer value with no decimal point. Otherwise it will be interpreted as an increment on the corresponding WCS axis. Thus, if the current WCS Frame of NDF "image" describes (RA,Dec) axes:

```
image(10:00:00~0:0:10, -24:00:00~0:10:00)
```

could be used to refer to region centred at (RA,Dec) (10:00:00,-24:00:00) and covering an extent of 10 seconds on the RA axis and 10 arc-minutes on the Dec axis. Note, the RA extent is not an arc-distance, it is an increment in Right Ascension value. So the corresponding arc-distance will depend on the declination and will become smaller as either pole is approached. However, it is possible to indicate that the extent is an arc-distance by appending one of the three string "as" (arc-seconds), "am" (arc-minutes) or "ad" (arc-degrees) to the end of the extent specifier. For instance:

```
image(10:00:00~10am, -24:00:00~50as)
```

will cause the section to cover an RA extent corresponding to 10 arc-minutes and a Dec extent corresponding to 50 arc-seconds. An error will be reported if the current WCS Frame is not suitable (i.e. is not a SkyFrame).

If the WCS axes are rotated, the NDF section actually used will be a box just large enough to hold the requested range of WCS axis values.

16.4 Using Normalised Pixel Coordinates to Specify Sections

If a numerical value representing a bound, central value, or extent is followed by a percent sign (%) then the numerical value is interpreted as a percentage of the full width of the NDF on the corresponding pixel axis. Thus, to specify the central 50% of an image on both pixel axes, either of the following sections could be used:

```
image( ~50%, ~ 50%)
      image( 25%:75%, 25%:75% )
```

16.5 Changing Dimensionality

The number of dimensions given when specifying an NDF section need not necessarily correspond with the actual number of NDF dimensions, although usually it will do so.

If fewer dimensions are specified than there are NDF dimensions, then any unspecified bounds will be set to (1:1) for the purposes of identifying the pixels to which the section should refer. Conversely, if extra dimensions are given, then the shape of the NDF will be padded with extra bounds set to (1:1) in order to match the number of dimensions. In all cases, the resulting section will have the number of dimensions actually specified, the padding serving only to identify the pixels to which the section should refer.

This process corresponds exactly to that which takes place via the programming interface when NDF_SECT is called (see §15.6).

16.6 Mixing Bounds Expressions

WCS (or *axis*) coordinates can be mixed with pixel indices in the same subscript expression. In fact, any of the features described earlier may be combined when specifying an NDF section, the only restrictions being:

(1) When the shape of the resulting section is expressed in pixel indices, the lower bound must not exceed the upper bound in any dimension.

(2) If the bounds for an axis are specified by centre and width values (rather than as lower and upper bounds), then a WCS value should not be used with a pixel index. That is, the centre and width values must both refer to the same coordinate system.

Thus, all the following might be used as valid specifications for NDF sections:

```
ndf(3.7)
ndf(,5:)
ndf(-77:12h22m12s,,4)
ndf(66~9,4:17)
ndf(~5,6~)
ndf(~,:)
ndf(5500.0~150,)
ndf(3.0~1.5,-78.06D-3:13.0545,,,)
```

Many other combinations are obviously possible. In cases where some bounds are given in pixel indices and some in WCS coordinates, two boxes will be formed; one representing the pixel index bounds and one representing the WCS bounds. The actual NDF section used will be the overlap of the two boxes. The pixel box will inherit any pixel index limits supplied in the bounds expression, and will use default values for any missing limits. These default pixel index bounds are just the bounds of the base NDF. Likewise, the WCS box will inherit any WCS limits supplied in the bounds expression, and will use default values for any missing limits. The default WCS limits are the bounds of a box that just includes the whole pixel box.

17 MERGING AND MATCHING NDF ATTRIBUTES

17.1 The Problem

A problem which immediately surfaces when you start to write applications to process NDFs is how to cope with the wide variety of data structures which may be encountered. Taking account of all possible shapes and sizes, all feasible types and storage forms for NDF components, the presence or absence of *bad* pixels and the state (defined or undefined) of each component, the number of combinations is clearly enormous. So how can simple general-purpose applications be written to cope with all this?

In simple cases (*e.g.* only a single input NDF) it is possible for an application to enquire about certain NDF attributes and adapt to some extent to take account of them. In the example in §9.4, for instance, separate algorithms were used depending on whether *bad* pixels might be present or not. However, even this modest degree of adaption can rapidly become complicated if there are two or more NDFs (and hence four or more combinations of *bad*-pixel flags) to consider.

For general-purpose software which will be heavily used, some attempt to adapt will probably be worthwhile if it leads to significantly better performance. However, it is often necessary to write very simple or "one off" applications with little knowledge of NDF data structures, where any need to adapt to an incoming NDF would impose an unwelcome programming burden. Such applications should nevertheless make a consistently good job of processing NDF data structures.

The problem, therefore, is how to reconcile the very diverse requirements imposed by the complicated nature of NDF data structures with the variable, but generally far more modest capabilities of real applications. The solution lies in various conversion processes which allow the attributes of NDFs to be *merged* and *matched* to the capabilities of applications, to arrive at a compromise method of processing the information in any particular set of NDF data structures. As this description suggests, some loss of efficiency or information will inevitably be involved, but this will usually be acceptably small. In cases where the penalty is unacceptable, an application naturally has the option of aborting with an appropriate error message.

17.2 Merging and Matching Bad-Pixel Flags

First consider processing the *data* arrays of two input NDFs. Under what circumstances should checks for *bad* pixels be made?

The capabilities of real applications in this area usually fall into one of the following categories:

(1) No ability to handle *bad* pixels at all.

(2) The ability to check all arrays for *bad* pixels, regardless of the *bad*-pixel flag(s).

(3) The ability to adapt the processing algorithm according to whether *bad*-pixels may be present or not.

It is expected that the majority of applications will fall into categories 1 and 2, since only a few general-purpose software items are likely to obtain worthwhile benefit from attempting to

optimise their behaviour when *bad* pixels are known to be absent (category 3). Nevertheless, it is instructive to consider this last and most complicated category first.

As already noted, there are four possible combinations of *bad*-pixel flags if we consider the *data* components of a pair of NDFs (there would be considerably more if we wanted to include the *variance* components as well). For all practical purposes, however, the number of combinations can be reduced to two by *merging* the *bad*-pixel flags of the separate array components using a logical "OR", for instance:

```
      LOGICAL BAD1, BAD2, BAD

      ...

      CALL NDF_BAD( INDF1, 'Data', .FALSE., BAD1, STATUS )
      CALL NDF_BAD( INDF2, 'Data', .FALSE., BAD2, STATUS )
      BAD = BAD1 .OR. BAD2

      ...
```

The resulting value of BAD could then be used to determine whether checks for *bad*-pixel values are necessary. For instance, if the two *data* arrays were to be added, the main processing algorithm might look like this:

```
      SUBROUTINE ADDIT( BAD, EL, A, B, C, STATUS )
      INCLUDE 'SAE_PAR'
      INCLUDE 'PRM_PAR'

      LOGICAL BAD
      INTEGER EL, STATUS, I
      REAL A( EL ), B( EL ), C( EL )

      IF ( STATUS .NE. SAI__OK ) RETURN

*  No bad-pixel checks needed, so add the arrays.
      IF ( .NOT. BAD ) THEN
         DO 1 I = 1, EL
            C( I ) = A( I ) + B( I )
 1       CONTINUE

*  Bad pixel checks needed, so check and assign a bad result if necessary.
      ELSE
         DO 2 I = 1, EL
            IF ( A( I ) .EQ. VAL__BADR .OR. B( I ) .EQ. VAL__BADR ) THEN
               C( I ) = VAL__BADR

*  Otherwise add the array elements normally.
            ELSE
               C( I ) = A( I ) + B( I )
            END IF
 2       CONTINUE
      END IF
      END
```

Although this is the most sophisticated response to the presence of *bad* pixels which we will consider, two compromises have nevertheless still been made here. First, the main processing algorithm will sometimes run slightly more slowly than is absolutely necessary, because taking the logical "OR" means that checks will occasionally be made for *bad* pixels in arrays which do not contain any. Secondly, as a result of this, some "good" pixels may accidentally be identified as *bad* and a small fraction of valid pixels might therefore be lost. In practice, both of these penalties are usually acceptably small, given the alternative (of writing many versions of the main processing algorithm to cater for every combination of *bad*-pixel flags).

So what should happen if the main processing algorithm is very simple and cannot handle any *bad* pixels at all (category 1 above)? In this case, it is not possible to compromise, because attempting to ignore the situation and process an array containing *bad* pixels without checking for them will give a completely wrong result (as well as causing severe numerical problems which will probably cause the application to crash). In this situation, the limitations of the application are paramount, and the correct response is to report an error and abort (see §9.5). A user would then have the option of running a separate application to "repair" the data by replacing *bad* pixels with an acceptable alternative.

The need to handle *bad*-pixel flags in any of the ways described above is very common, so the prototype *merging and matching* routine NDF_MBAD is provided to simplify the process. This routine merges the *bad*-pixel flags of a pair of NDFs and matches them to the capabilities of an application. Its first (BADOK) argument is supplied with a logical value indicating whether the calling application can handle *bad* pixels or not, for instance:

```
LOGICAL BADOK, CHECK, BAD

...

CALL NDF_MBAD( BADOK, INDF1, INDF2, 'Data', CHECK, BAD, STATUS )
```

If the BADOK argument is set to .TRUE., then NDF_MBAD simply returns the logical "OR" of the *bad*-pixel flags for each of the two NDFs via its BAD argument. In this case it behaves in most respects exactly like its simpler relative NDF_BAD (see §9.4). However, if BADOK is set to .FALSE., but the returned BAD value would nevertheless be .TRUE. (indicating that *bad* pixels may be present in one of the NDFs), then an error results. NDF_MBAD then makes an appropriate error report, explaining that the application cannot handle the *bad* pixels, and returns with its STATUS argument set to NDF__BADNS (bad values not supported), as defined in the include file NDF_ERR. In the normal course of events, the application would then abort and this message would be delivered to the user.

A call to NDF_MBAD therefore allows an application to declare whether it can handle *bad* pixels, and then determines whether checks for *bad* pixels should actually be made. At the same time, it will handle any problems which may arise.

If more than two NDF's are involved, then the routine NDF_MBADN may be used instead of NDF_MBAD. This routine exists for merging the *bad*-pixel flag values of an arbitrary number of NDFs, whose identifiers are supplied in an integer array. For instance:

```
INTEGER N, NDFS( N )

...
```

```
CALL NDF_MBADN( .FALSE., N, NDFS, 'Data', .TRUE., BAD, STATUS )
```

could be used by an application which was incapable of handling *bad* pixels in order to check for their existence in a whole sequence of NDFs at once.

Finally, if only a single NDF is being used, then there is a choice. The routine NDF_MBADN may be called with N set to 1, but NDF_MBAD may also be used by passing the same identifier value twice, as follows:

```
CALL NDF_MBAD( BADOK, INDF1, INDF1, COMP, CHECK, BAD, STATUS )
```

NDF_MBAD is designed to expect this kind of usage, and will detect it and behave appropriately without any loss of efficiency. All the other equivalent matching and merging routines described in subsequent sections also have this capability.

17.3 Matching NDF Bounds

The common situation where an application takes two or more input NDFs and combines their array component values pixel-by-pixel (*e.g.* to multiply their *data* arrays together) raises a similar problem of how to cope with the diversity of NDF shapes which could be encountered. In this case the capabilities of the application are generally not an issue, because most applications will be written to process NDFs of arbitrary size. However, what steps should be taken if the NDFs supplied as input turn out to have different sizes?

A simple approach might be to check whether the shapes of the input NDFs match and to report an error and abort if they do not. This straightforward (but unfriendly) approach requires a significant number of Fortran statements and quickly becomes a chore if it has to be performed explicitly in every application. However, by making use of NDF *sections* (see §15.2) to change the apparent pixel-index bounds of an NDF, it is possible to do considerably better than this, and to allow processing of NDFs whose bounds may not initially match.

To understand the principle, consider two NDFs with shapes which do not match, such as:

(15:115,-5,10)

and

(1:200,1:30)

It would clearly be meaningless to attempt to multiply the *data* arrays of these two NDFs together pixel-by-pixel as they stand, since there are some pixels in the first one which do not exist in the second one, and *vice versa*. However, the pixels within the region:

(15:115,1:10)

do exist within both NDFs and can be multiplied together if we decide to discard the rest. This can be done by creating an appropriate section from each NDF which selects only these common pixels for further processing. This task, of calculating which pixels to use and of selecting the appropriate NDF sections, may be performed using the routine NDF_MBND, which matches the bounds of a pair of NDFs, as follows:

```
CALL NDF_MBND( 'TRIM', INDF1, INDF2, STATUS )
```

If necessary, NDF_MBND will annul the NDF identifiers supplied and replace them with identifiers to appropriate NDF sections with matching shapes. The array components of these sections may then be accessed and compared pixel-by-pixel, since they will be of equal size. If the two NDFs have no pixels in common, then NDF_MBND will report an appropriate error message and set a STATUS value.

In cases where more than two NDFs are involved, the similar routine NDF_MBNDN may be used. This will match the bounds of an arbitrary number of NDFs, whose identifiers should be held in an integer array, for instance:

```
INTEGER N, NDFS( N )

...

CALL NDF_MBNDN( 'TRIM', N, NDFS, STATUS )
```

This routine will create sections with matching bounds from each NDF supplied, selecting only those pixels which exist in all of the NDFs. Once again, an appropriate error report will be made, and a STATUS value set, if this cannot be achieved.

If access to any of the original NDFs is still required, then the routine NDF_CLONE may be used to retain an identifier for it before calling NDF_MBND or NDF_MBNDN, since either of these latter routines may annul the identifiers passed to them. For example:

```
CALL NDF_CLONE( INDF1, INDFK, STATUS )
CALL NDF_MBND( 'TRIM', INDF1, INDF2, STATUS )
```

would ensure that an identifier INDFK is retained for the NDF whose original identifier was held in INDF1, even although NDF_MBND may alter the INDF1 variable so that it refers only to a sub-section of the NDF.

An alternative method of matching NDF bounds is by *padding*, in which each NDF is padded out with *bad* pixels rather than by discarding pixels at the edges. It may be employed by specifying 'PAD' for the first (OPTION) argument to NDF_MBND or NDF_MBNDN. For instance:

```
CALL NDF_MBND( 'PAD', INDF1, INDF2, STATUS )
```

In this case, the NDF sections produced may represent super-sets of the original NDFs (see §15.4). This option is needed less often, but it has the advantage that no pixels will be lost. It is generally used in cases where an output pixel should be assigned a value even if one of the input pixels is missing (or has a *bad* value). For instance, an application to merge the *data* arrays of two non-congruent NDFs by averaging their pixel values might use the 'PAD' option to match the bounds of the input NDFs. It could then generate an output array using an algorithm along the following lines, where an output pixel value is generated if either or both input pixels are valid:

```
INTEGER EL, I
REAL A( EL ), B( EL ), C( EL )
INCLUDE 'PRM_PAR'
```

```
      ...

*  Loop to process each array element.
      DO 1 I = 1, EL

*  If both input pixels are good, then take the average.
         IF ( A( I ) .NE. VAL__BADR .AND. B( I ) .NE. VAL__BADR ) THEN
            C( I ) = 0.5 * ( A( I ) + B( I ) )

*  Otherwise, if the first one is good, then use it.
         ELSE IF ( A( I ) .NE. VAL__BADR ) THEN
            C( I ) = A( I )

*  Otherwise use the second one.
         ELSE
            C( I ) = B( I )
         END IF
 1    CONTINUE
```

When using the 'TRIM' option, the output NDF may be smaller than either of the two input NDFs, and when using the 'PAD' option, it may be larger. However, the application usually need not be aware of this.

17.4 Merging and Matching Numeric Types

The variety of possible numeric types which may be used to store the values of NDF array components (see §7.1) poses another problem for the applications programmer. What type of arithmetic should be chosen to process the values?

The simplest option, and the one which is recommended for normal use, is:

> *Use single-precision floating-point arithmetic for calculations wherever possible...*

and access all array components (other than *quality*) as '_REAL' values, taking advantage of the implicit type conversion provided by the NDF_ routines if necessary (see §8.7).

However, when writing general-purpose software which may be heavily used, the possibility of duplicating the main processing algorithm so that calculations can be performed using several alternative types of arithmetic might also be considered. This allows applications to support the processing of unusual numeric types (*e.g.* double-precision), while normally using less computationally expensive arithmetic (*e.g.* single-precision) when that is adequate. The ability to explicitly handle values with a variety of numeric types also makes it less likely that an expensive type conversion will have to be performed implicitly by the NDF_ routines (see §8.7). Of course, the disadvantages of this approach are that extra work is involved in duplicating the main processing algorithm. The magnitude of this extra work should not be underestimated, although the use of the GENERIC compiler (SUN/7) can simplify the task to some extent.

To give a concrete example, suppose you decide to duplicate the main processing algorithm in an application so that it can perform calculations using either single- or double-precision arithmetic. Input NDFs with numeric array types of '_REAL' or '_DOUBLE' can then be processed directly, although with other numeric types (or with mixed types) implicit type conversion will still

need to occur. This can get rather complicated to sort out, so a simple routine NDF_MTYPE is provided to make the decision about which version of an algorithm to invoke in a particular case, depending on the declared capabilities of an application, for instance:

```
      INCLUDE 'NDF_PAR'
      CHARACTER ITYPE * ( NDF__SZTYP ), DTYPE * ( NDF__SZFTP )

      ...

      CALL NDF_MTYPE( '_REAL,_DOUBLE', INDF1, INDF2, 'Data', ITYPE, DTYPE, STATUS )
```

The first (TYPLST) argument is a list of the numeric types which the application can process explicitly ('_REAL' and '_DOUBLE') supplied in order of preference, *i.e.* in order of increasing computational cost in this instance. NDF_MTYPE will examine the two NDFs supplied and select a numeric type from this list to which the *data* component values should be converted for processing so that no unnecessary loss of information occurs. The conversion from '_REAL' to '_DOUBLE' would be acceptable, for instance, whereas the opposite conversion process would lose information.

The resulting *implementation type* is returned as an upper-case character string via the ITYPE argument, and may be used both as an input argument to NDF_MAP for accessing the NDFs' values, as well as for deciding which version of the main algorithm to invoke, for instance:

```
      CALL NDF_MTYPE( '_REAL,_DOUBLE', INDF1, INDF2, 'Data', ITYPE, DTYPE, STATUS )

*  Access the input array values using the selected implementation type.
      CALL NDF_MAP( INDF1, 'Data', ITYPE, PNTR1, EL, STATUS )
      CALL NDF_MAP( INDF2, 'Data', ITYPE, PNTR2, EL, STATUS )

*  Invoke the appropriate version of the processing algorithm.
      IF ( ITYPE .EQ. '_REAL' ) THEN
         <invoke the single-precision algorithm>
      ELSE IF ( ITYPE .EQ. '_DOUBLE' ) THEN
         <invoke the double-precision algorithm>
      END IF
```

The NDF_MTYPE routine also addresses the question of how to store the results of the calculation since, ideally, the numeric type used for the output array(s) should preserve the precision of the calculation without unnecessarily increasing the storage space required. A suitable numeric type is therefore returned as an upper-case character string via the DTYPE argument and may be used as an input argument to a routine such as NDF_CREAT which creates an output NDF. More commonly, however, it will be used to change the numeric type of an NDF which has already been created by a call to NDF_PROP (see §14.4), for instance:

```
      CALL NDF_PROP( INDF1, ' ', 'OUT', INDF3, STATUS )
      CALL NDF_MTYPE( '_REAL,_DOUBLE', INDF1, INDF2, 'Dat,Var', ITYPE, DTYPE, STATUS )
      CALL NDF_STYPE( DTYPE, INDF3, 'Dat,Var', STATUS )
```

Here, a new output NDF is created based on the first input NDF by using NDF_PROP, and an identifier INDF3 is obtained for it. NDF_MTYPE is then called to determine an appropriate numeric type for storing the output *data* and *variance* components, and NDF_STYPE is invoked

to set the output components' type appropriately. In a typical application, the input and output arrays would probably then be accessed and passed to the appropriate version of the main processing algorithm. A complete example of this sequence of events can be found in §A.7, where its integration with the handling of the *bad*-pixel flag and matching of the NDFs' bounds is also demonstrated.

Note that an equivalent routine NDF_MTYPN also exists for merging and matching the numeric types of an arbitrary number of NDFs, whose identifiers are supplied in an integer array, as follows:

```
INTEGER N, NDFS( N )

...

CALL NDF_MTYPN( '_INTEGER,_REAL', N, NDFS, 'Data', ITYPE, DTYPE, STATUS )
```

Both this routine and NDF_MTYPE will report an error and set STATUS to NDF__TYPNI (processing of data type not implemented), as defined in the include file NDF_ERR, if it is not possible to select an implementation type from the list supplied without leading to loss of information. This would be the case for instance, if one of the NDFs had a *data* component with a numeric type of '_DOUBLE' but the application could only perform single-precision arithmetic. In this case, the values returned via the ITYPE and DTYPE arguments would be the "best compromise" values and could still be used, if required, by ignoring the error condition. For instance:

```
INCLUDE 'NDF_ERR'

...

CALL ERR_MARK
CALL NDF_MTYPE( '_REAL', INDF1, INDF1, 'Data', ITYPE, DTYPE, STATUS )
IF ( STATUS .EQ. NDF__TYPNI ) CALL ERR_FLUSH( STATUS )
CALL ERR_RLSE
```

Here, the error system routine ERR_FLUSH has been used to deliver the error message to the user and reset STATUS (see SUN/104). The user would then receive a warning message explaining that '_DOUBLE' data could not be handled without loss of information, and that conversion to '_REAL' would have to be used. The application could then perform the conversion and processing, having warned the user about the unavoidable loss if information which will occur.

18 THE AXIS COORDINATE SYSTEM

This section describes the concepts which an NDF's *axis* coordinate system represents. The following section (§19) then goes on to consider how to access an NDF's *axis* components, which hold information about this coordinate system.

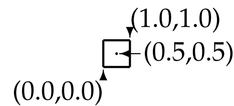
18.1 Pixel Coordinates

Hitherto, an NDF has been considered simply as an N-dimensional array of pixels, addressed by a set of pixel indices. Since they are integer quantities, these indices cannot represent a continuous coordinate system, although the information stored in an NDF will almost always require that positions within it be describable to sub-pixel accuracy. For example, a calculation to determine the centroid position of a star in a 2-dimensional image will inevitably give rise to a non-integer result, for which a continuous (x, y) coordinate system will be required.

There are a number of ways in which a continuous coordinate system can be defined for a regular array of pixels. In the absence of other information, the NDF convention is to use a *pixel coordinate system* in which a pixel with indices (i, j) has its centre at the position:

$$(i - \tfrac{1}{2}, j - \tfrac{1}{2})$$

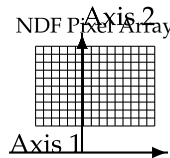
and is taken to be one unit in extent in each dimension. Pixel (1,1) would therefore be centred at the position (0.5,0.5) and would have its "lower" and "upper" corners located at positions (0.0,0.0) and (1.0,1.0) respectively, as follows:



This makes it possible to refer to fractional pixel positions—in this case within a 2-dimensional array, although the principle can obviously be extended to other numbers of dimensions.

18.2 Axis Coordinates

The pixel coordinate system described above defines how to convert pixel indices into a set of continuous coordinates and therefore introduces a coordinate *axis* which runs along each dimension of the NDF, as follows:



The use of the pixel size to determine the units of these axes is rather restrictive, however, and in practice we may want to use more realistic physical units. This would allow a spectrum to be calibrated in wavelength, for instance, or the output from a plate-measuring machine to be related to axes calibrated in microns.

Of course, the pixel coordinate system is only the default choice, and is intended to be used only in the absence of other information. The NDF's *axis* components are designed to hold the extra information needed to define more useful coordinate systems, so that realistic axes can be associated with a NDF, along with *labels* and *units* for these axes. The method used also allows for the possibility that an NDF's pixels may not be square and that they may not be contiguous (*i.e.* that they may have gaps between them, or may overlap) when their positions are expressed in *axis* units. Statistical uncertainty in the pixel positions may also be represented, if present.

18.3 Axis Arrays

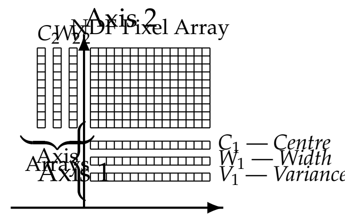
To define the pixel coordinate system in §18.1, we specified the location of each pixel by giving its *centre* position and *width* on each *axis*. Thus, for a given dimension, the pixel *centre* position C was derived from the corresponding pixel index *i* according to the formula:

$$C(i) = i - \frac{1}{2}$$

and its *width* W was given by:

$$W(i) = 1$$

An NDF's *axis* coordinate system extends this idea by allowing each of these *centre* and *width* functions to be determined by values stored in a 1-dimensional array. These *axis arrays* then act as "look-up tables" which convert pixel indices into pixel *centre* coordinates and *width* values on each *axis:*



This allows a wide range of possible coordinate systems to be accommodated. A third *axis variance* array is also provided as a look-up table to convert pixel indices into *variance* estimates, which can be used to represent any possible statistical uncertainty in a pixel's *centre* position.

18.4 Pixel Positions and Dimensions

If C_n and W_n represent the *axis centre* and *width* arrays for the n'th dimension of an NDF, then a pixel with index i in this dimension has its *centre* at coordinate C_n(i) and has a *width* of W_n(i) on the corresponding *axis*. It therefore extends along the *axis* from the point:

$$C_n(i) - \frac{1}{2}W_n(i)$$

to the point:

$$C_n(i) + \frac{1}{2}W_n(i)$$

In two dimensions the central (x, y) coordinate of a pixel with indices (i, j) would therefore be given by:

$$(x,\ y) = (\ C_1(i),\ C_2(j)\ )$$

and its size would be:

$$\Delta x\ \times\ \Delta y = W_1(i)\ \times\ W_2(j)$$

The *axis variance* array is used to represent any statistical uncertainty in a pixel's *centre* position and hence in the position of the pixel as a whole.[16] Like the NDF's main *variance* component (§8.9), its values are estimates of the mean squared error in the pixel's position, so the value which would normally be quoted as the positional uncertainly (or used to plot error bars) is the square root of this value. *Axis variance* arrays may also be accessed directly as standard deviation values if required (see §19.10).

18.5 Default Axis Array Values

An important feature of each *axis* array is a set of default values which serve to define the *axis* coordinate system in the absence of complete information. In the simplest case (*i.e.* no information), this reduces to the pixel coordinate system discussed in §18.1. The following describes how default values are obtained for each *axis* array:

Centre: If values are required for an *axis centre* array and none have been provided, then its values are set equal to $i - \frac{1}{2}$, where i is the pixel's index in the relevant dimension. Thus, if an NDF had pixel-index bounds (3:10) in a particular dimension, the default *axis centre* array values for this dimension would be:

2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5, 9.5

Width: If values are required for an *axis width* array and none have been defined, then its values are derived from the corresponding *axis centre* array by forming differences between the *centre* coordinates of the neighbouring pixels, *i.e.* the default *width* values are obtained as follows:[17]

$$W_n(i) = \frac{1}{2}|C_n(i+1) - C_n(i-1)|$$

This means that the default pixel *widths* match the local average spacing between pixel *centres*, which is usually appropriate. Note, however, that this does not guarantee that the pixels will be contiguous (*i.e.* that their edges will meet exactly) except in cases where the pixel *centres* are uniformly spaced (see §18.6).

Variance: If no *axis variance* array values have been defined, then they default to zero, implying no uncertainty in the pixel *centre* positions.

18.6 Contiguous and Non-Contiguous Pixels

It is important to note that the meanings attached to an NDF's *axis* arrays are defined rather precisely. In particular, note that the *axis centre* array specifies the position of the geometrical *centre* of a pixel, *i.e.* the mid-point between its edges, so that a pixel will always extend by an equal amount on either side of this position.

The edges of adjacent pixels will therefore only meet exactly (*i.e.* there will be no overlap or gap) if their *centre* and *width* values are related in the correct way. To be precise, adjacent pixels with

[16] There is no corresponding provision for recording any uncertainty in a pixel's *width*.

[17] If either of the neighbouring *centre* values does not exist (because the pixel is at the end of the array) then it is replaced by $C_n(i)$ and the $\frac{1}{2}$ in the formula is dropped. If neither neighbour exists (because the NDF's dimension size is 1) then the *width* value is set to unity. Note that the default *centre* array values will be used if none have been defined, and this will also result in *width* values of unity.

indices i and $i+1$ must have *centre* positions separated by half the sum of their *widths* if they are to be contiguous along a particular *axis*, so that:

$$|C_n(i+1) - C_n(i)| = \frac{1}{2}[W_n(i+1) + W_n(i)]$$

With contiguous pixels (the normal case), this means that the *axis centre* and *width* arrays are not independent. In fact, either could be derived from the other to within a constant, but since this constant cannot be found without additional information, it is often necessary to store both arrays. However, an important exception occurs if the pixel *centres* are evenly spaced, because a convenient method then exists of deriving the *width* array from the *centre* array so that contiguous pixels always result. This is the method used to generate default *axis width* values when necessary (§18.5).

To avoid any potential ambiguity about the interpretation of *axis* array values and whether an NDF's pixels should be considered contiguous or not, the following recommendations are given about the information which should be stored in *axis* arrays:

- If the default NDF pixel coordinate system is satisfactory, then it should be used and no *axis* coordinate information should be defined. In this case the pixels will always be contiguous.
- Otherwise, if the pixel *centres* are evenly spaced, then...
  - If the pixels are contiguous, the *axis centre* array should be assigned values but the associated *width* array may be left undefined.
  - If the pixels are not contiguous, both the *axis centre* and *width* arrays should be assigned values.
- Otherwise, if the pixel *centres* are un-evenly spaced, then both the *axis centre* and *width* arrays should always be assigned values.

18.7 Processing Axis Array Values

The method by which NDF *axis* arrays are processed should also reflect their meanings, as defined above. By way of illustration, suppose that a transformation of *axis* values is to be performed, so that each *axis* coordinate is converted to a new value by means of some non-linear function. Rather than simply applying this function to calculate new pixel *centre* locations from the old ones, the correct procedure is to transform the pixel edge locations and to derive new *centre* positions from these, as follows:

(1) Obtain the relevant pixel *centre* and *width* arrays, accepting their default values if necessary.

(2) From these, calculate the positions of the edges of each pixel.

(3) Transform the edge positions using the non-linear transformation function.

(4) Calculate new pixel *centre* positions (mid-way between the new edge positions) and store them in the NDF's *axis centre* array.

(5) Calculate associated pixel *width* values (from the difference in the pixel edge positions). Since the pixel *centres* will now be non-uniformly separated, these new *width* values should also be stored in the NDF's *axis width* array.

(6) If *axis variance* array values are available, then these should be propagated through the transformation function using the usual error-propagation formulae.

This procedure is necessary to ensure that the pixels remain contiguous (or non-contiguous, if appropriate) and that the new *centre* positions lie mid-way between the new pixel edge locations. Furthermore, the operations above can all be reversed if necessary to recover the original *axis* array values.

18.8 Axis Normalisation

One aspect of the *axis* coordinate system which has not yet been discussed is the property of *axis normalisation*, which is indicated by a logical *normalisation flag* associated with each *axis*. This flag does not affect the interpretation of the *axis* information itself, but instead determines how the NDF's *data* and *variance* arrays should behave when the associated *axis* information is modified.

If the normalisation flag for an NDF *axis* is set to .TRUE., then it indicates that the NDF's *data* values (and by implication its *variance* values) are *normalised* to the pixel *width* values for that *axis*. To give an example, suppose that a spectrum contains *data* values representing energy accumulated per unit of wavelength, with each pixel having a known spread in wavelength. In this case, the sum of each pixel's *data* value multiplied by its *width* will give the total energy in any part of the spectrum. This is an important property which may need to be retained if the *axis width* values are altered for any reason (*e.g.* to apply an instrumental correction, or to allow for red-shift).

The *axis* normalisation flag indicates whether this type of normalisation should be preserved. If it is set to .TRUE., and the associated *axis width* values are modified, then each NDF *data* value should be multiplied by an appropriate factor so that its *data* $\times$ *width* product remains unchanged. If present, the *variance* values should also be corrected by multiplying by the square of this factor. In cases where more than one *axis* normalisation flag is set to .TRUE., the correction factors for each *axis* must be applied in turn.

If all the *axis* normalisation flags are set to .FALSE. (the default situation), then no changes to the *data* or *variance* components will be necessary if the *axis width* values are modified.

19 AXIS COMPONENTS

19.1 Overview of an NDF's Axis Components

Information about an NDF's *axis* coordinate system is stored in its *axis components*, which are conveniently categorised as follows:

Axis character components:	LABEL	— Axis labels
	UNITS	— Axis units
Axis array components:	CENTRE	— Pixel centre coordinates
	WIDTH	— Pixel width values
	VARIANCE	— Variance estimates for pixel positions

As with the main components of an NDF, the names of these *axis* components are significant,[18] since they are used by the NDF_ routines to identify the component(s) to which certain operations should be applied. *Axis* component names are specified in the same way as those of the main components of an NDF, including the use of abbreviations, mixed case and comma-separated lists where appropriate (see §5.1 for details).

Access to an *axis* component must also specify the number of the NDF *axis* to be used. This is normally an integer lying between 1 and the number of NDF dimensions, but many routines will also accept a value of zero, indicating that an operation is to be applied to all of an NDF's *axes*. This additional item of information means that a separate set of routines must be provided for accessing *axis* components. Nevertheless, many of the principles described in earlier sections for accessing other NDF components are also applicable here, so the descriptions given below are relatively brief. References to more complete descriptions are given where appropriate.

The following describes the purpose and interpretation of each *axis* component in slightly more detail.

Axis Character Components:

LABEL – This is a character string, whose value is intended for general use for such things as labelling the axes of graphs or as a heading for columns in tabulated output; *e.g.* 'Scanner X offset'. There is a separate *axis label* value for each NDF dimension.

UNITS – This is a character string, whose value describes the physical *units* of the quantity measured along an NDF's *axis; e.g.* 'micron'. There is a separate *axis units* value for each NDF dimension.

Axis Array Components:

CENTRE – This is a 1-dimensional array which holds the coordinates of the pixel *centres* as described in §18.3. The values in this array should either increase or decrease monotonically with position in the array. There is a separate 1-dimensional *axis centre* array for each dimension of an NDF, whose size matches the size of the corresponding NDF dimension.

WIDTH – This is a 1-dimensional array which holds a set of non-negative *width* values for the NDF's pixels as described in §18.3. There is a separate 1-dimensional *axis width* array for each dimension of an NDF, whose size matches the size of the corresponding NDF dimension. These *width* values should be such that no point can lie inside more than two NDF pixels simultaneously (*i.e.* although pixels are allowed to overlap with their neighbours, they may not overlap with more distant pixels).

[18] Note that the name "CENTRE" used by the NDF_ routines to refer to an NDF's *centre* component differs from the actual name of the HDS object in which it is stored, which is "DATA_ARRAY".

VARIANCE – This is a 1-dimensional array which holds a set of non-negative *variance* estimates representing any statistical uncertainty in the value of the corresponding pixel *centre* coordinate, as described in §18.3. There is a separate 1-dimensional *axis variance* array for each dimension of an NDF, whose size matches the size of the corresponding NDF dimension.

19.2 Axis Component States

Like all NDF components, each *axis* component has a logical *state* attribute associated with it which indicates whether or not it has a previously-assigned value. The state of an *axis* component may be determined by using the routine NDF_ASTAT, specifying the component name and the number of the *axis* about which information is required, as follows:

```
INTEGER IAXIS
LOGICAL STATE

...

CALL NDF_ASTAT( INDF, 'Width', IAXIS, STATE, STATUS )
```

In this example, a .TRUE. result would be returned via the logical STATE argument if values had previously been assigned to the *width* array of the *axis* identified by the IAXIS argument.

Unlike other NDF components, no error will result from attempting to read the value of an *axis* component while it is in an undefined state. This is because the NDF_ system will always supply a default value if necessary. Thus, an *axis* component's state merely serves to indicate whether a pre-assigned value will be used, as opposed to an internally-generated default.

NDF_ASTAT will also accept a list of *axis* component names and will return the logical "AND" of the results for all the specified components. An IAXIS value of zero may also be supplied to indicate that all the NDF's axes should be considered at the same time. Thus, the single call:

```
CALL NDF_ASTAT( INDF, 'Label,Units', 0, STATE, STATUS )
```

could be used to determine whether all the NDF's axes had previously-assigned values for both their *label* and *units* components.

19.3 Restrictions on Axis Component States

In general, the logical state attribute of each *axis* component is independent and may be manipulated freely. However, there is one notable and important exception to this:

If any axis *component is to be in a defined state, then the* centre *arrays for* *all* *the NDF's* axes *must also be defined*

Thus, all an NDF's *axis centre* arrays behave as a single unit, and it is a pre-requisite that all of these arrays should be in a defined state before any other *axis* component may be assigned a value.

Of course, it would be very inconvenient if values had to be explicitly generated and assigned to all the *axis centre* arrays before any other *axis* values could be defined, so the the NDF_ system has an implicit mechanism for assigning default values to the *axis centre* arrays whenever they are required (*i.e.* whenever any *axis* component is assigned a value but the *axis centre* arrays are still undefined). A routine is also provided to perform this task explicitly if required, and is described in the next section.

If an NDF's *axis centre* arrays are in a defined state, then the *axis coordinate system* of the NDF as a whole is regarded as being defined. Otherwise (*i.e.* if no *axis* components are defined at all), then the *axis* coordinate system is undefined. The routine NDF_STATE can be used to test whether or not an *axis* coordinate system is defined by using a component name of 'Axis', thus:

```
CALL NDF_STATE( INDF, 'Axis', STATE, STATUS )
```

A .TRUE. value will be returned via the logical STATE argument if the *axis* coordinate system is defined.

19.4 Defining a Default Axis Coordinate System

The routine NDF_ACRE is provided to assign default values to all of an NDF's *axis* arrays, thereby defining a default *axis* coordinate system, as follows:

```
CALL NDF_ACRE( INDF, STATUS )
```

If the *axis* coordinate system is already defined, then all the *axis centre* arrays will already have values, so this routine will return without action. However, if this coordinate system (and hence each *axis centre* array) is undefined, then NDF_ACRE will assign default values to all the *centre* arrays, effectively defining a default *axis* coordinate system which is equal to the NDF's pixel coordinate system (§18.1).

After this operation, the values available from any of the NDF's *axis* components will be unchanged. This is because the NDF_ system would provide these same *axis centre* values as defaults in any case. Nevertheless, the definition of a default *axis* coordinate system is a significant step because it effectively takes a "copy" of the current pixel coordinate system. Any operation which subsequently changes the NDF's pixel indices (*e.g.* the application of pixel-index shifts – see §21.2) cannot then affect the pixel *centre* values, whereas previously it would have done.

19.5 Resetting Axis Components

The *axis* coordinate system of an NDF may be reset by specifying the component name 'Axis' in a call to NDF_RESET, as follows:

```
CALL NDF_RESET( INDF, 'Axis', STATUS )
```

This will cause all the NDF's *axis* components to become undefined, so that subsequent attempts to read values from any of them will return default values appropriate to the NDF's pixel coordinate system. A subsequent enquiry about the state of the 'Axis' coordinate system using NDF_STATE would return a value of .FALSE..

The routine NDF_AREST is also provided for resetting individual *axis* components. Thus, a particular *axis variance* array could be reset as follows:

```
CALL NDF_AREST( INDF, 'Variance', IAXIS, STATUS )
```

and any subsequent attempt to read from it would result in the default values being returned.

A list of component names may also be supplied to NDF_AREST, along with an optional IAXIS value of zero to indicate that the resetting operation should apply to all the NDF's axes at once. Thus, the following call:

```
CALL NDF_AREST( INDF, 'Wid,Var', 0, STATUS )
```

would reset all of an NDF's *axis width* and *variance* arrays.

Note that a component name of 'Centre' may not be supplied to NDF_AREST because all the *axis centre* arrays behave as a single unit and cannot be independently reset. The only way to remove all *axis centre* information from an NDF is to reset the entire *axis* coordinate system via a call to NDF_RESET, as above.

19.6 Accessing Axis Character Components

The *axis label* and *units* components of an NDF are both accessed via the same set of routines which behave in a similar manner to those for accessing an NDF's main character components (§6).

The value of either of these *axis* components may be obtained by calling the routine NDF_ACGET, as follows:

```
CHARACTER * ( 80 ) VALUE

...

VALUE = 'Default label'
CALL NDF_ACGET( INDF, 'Label', IAXIS, VALUE, STATUS )
```

This will return the value of the specified component, if it is defined. If its value is not defined and a non-blank default value has been set for the VALUE argument beforehand (as here), then this default value will be returned unchanged. However, if a blank VALUE string is provided, then the routine will generate its own default if necessary, returning either the value 'Axis n' for the *axis label* component (where n is the axis number) or 'pixel' for the *axis units*.

If an *axis* character component value is to be used in constructing a message, then it may be assigned directly to an MSG_ message token by means of the NDF_ACMSG routine. Thus, a message showing the *label* and *units* values for a particular NDF *axis* could be generated as follows:

```
CALL NDF_ACMSG( 'LABEL', INDF, 'Label', IAXIS, STATUS )
CALL NDF_ACMSG( 'UNITS', INDF, 'Units', IAXIS, STATUS )
CALL MSG_OUT( 'MESSAGE', '^LABEL (^UNITS)', STATUS )
```

Here, 'LABEL' and 'UNITS' are the names of message tokens (see SUN/104).

New values may be assigned to *axis* character components by using the routine NDF_ACPUT. For instance:

```
CALL NDF_ACPUT( 'Wavelength', INDF, 'Lab', 1, STATUS )
CALL NDF_ACPUT( 'nm', INDF, 'Unit', 1, STATUS )
```

would assign the *label* value 'Wavelength' and the *units* value 'nm' to *axis* 1 of an NDF. Note that the entire character string will be assigned (including trailing blanks if present) and the length of the component will be adjusted to match the new value. A value of zero for the fourth (IAXIS) argument would cause the value to be assigned to all of the NDF's axes.

After a successful call to NDF_ACPUT, the *axis* character component's state becomes defined. So, also, does the NDF's *axis* coordinate system—this means that default values will be assigned to all the NDF's *axis centre* arrays if these were not previously defined. The effect of this is exactly the same as if the routine NDF_ACRE (§19.4) had been called immediately before the call to NDF_ACPUT.

The length of an *axis* character component (*i.e.* the number of characters it contains) is determined by the last assignment made to it, (*e.g.* by NDF_ACPUT) and may be obtained using the routine NDF_ACLEN. For instance:

```
INTEGER LENGTH

...

CALL NDF_ACLEN( INDF, 'Units', IAXIS, LENGTH, STATUS )
```

will return the number of characters in the specified *axis units* component via the LENGTH argument. If the component is in an undefined state, then NDF_ACLEN will return the number of characters in the default value which would be returned by NDF_ACGET under these circumstances (see §19.6).

19.7 Mapping Axis Arrays for Reading

Access to *axis* array components takes place in much the same way as access to the main array components of an NDF. It also depends on the concept of *mapping*, as described in §8.

The routine NDF_AMAP provides mapped access to an *axis* array. Thus, to read values from an NDF's *axis centre* array, the following call might be used:

```
INTEGER PNTR( 1 ), EL

...

CALL NDF_AMAP( INDF, 'Centre', IAXIS, '_REAL', 'READ', PNTR, EL, STATUS )
```

Here, a numeric type of '_REAL' has been specified to indicate that an array of single-precision values is required and the mapping mode of 'READ' indicates that values are to be read, but not modified. The routine returns an integer pointer to the mapped values via its PNTR argument

and a count of the number of elements mapped via its EL argument (PNTR is actually a 1-dimensional array, so the pointer value in this example will be returned in its first element). The value returned for EL will be equal to the size of the NDF dimension being accessed.[19] The mapped values may be accessed in the normal way by passing them to a subroutine using the %VAL facility (see §8.2).

If the *axis* array being accessed is in an undefined state, then a set of default values will be returned (see §18.5). Note that the mapping mode *initialisation options* available when mapping the main NDF array components (§8.6) cannot be applied to *axis* arrays.

NDF_AMAP will also accept a list of *axis* component names and will map all of them in the same way (*i.e.* with the same type and mapping mode). Pointers to each mapped array will be returned via the PNTR argument, which must have sufficient elements to accommodate the returned values. The following example shows how access to all the *axis* arrays for a particular NDF dimension could be obtained using this facility, and then passed to another routine for processing:

```
      INTEGER PNTR( 3 ), EL

      ...

      CALL NDF_AMAP( INDF, 'Cent,Width,Var', IAXIS, '_DOUBLE', 'READ', PNTR, EL,
     :               STATUS )
      CALL DOAXIS( EL, %VAL( PNTR( 1 ) ), %VAL( PNTR( 2 ) ), %VAL( PNTR( 3 ) ),
     :             STATUS )
```

Note that it is not possible to map *axis* arrays for all the *axes* of an NDF in a single call to NDF_AMAP, because each would require a different value of EL to be returned. An IAXIS value of zero is therefore not permitted when calling NDF_AMAP.

19.8 Unmapping Axis Arrays

When access to an *axis* array is complete, it should be *unmapped* in the usual way (see §8.2). There are a number of methods by which this can be done. Most simply, the cleaning-up action of NDF_END (§3.4) may be relied upon to annul an NDF identifier and to unmap any mapped arrays associated with it as part of this process (§8.3). This will correctly deal with any *axis* arrays which may be mapped.

Alternatively, the routine NDF_UNMAP may be used by specifying a component name of 'Axis', as follows:

```
      CALL NDF_UNMAP( INDF, 'Axis', STATUS )
```

This will unmap all *axis* arrays which are mapped. (A "wild-card" component name of '∗' will also affect *axis* arrays in the same way.)

If *axis* arrays are to be unmapped individually, then the routine NDF_AUNMP should be used. Thus,

```
      CALL NDF_AUNMP( INDF, 'Width', IAXIS, STATUS )
```

[19]Under error conditions a "safe" value of 1 will be returned for EL, as discussed in §4.4.

might be used to unmap a particular *axis width* array. As usual, a list of *axis* component names may be supplied to NDF_AUNMP and a "wild-card" *axis* component name of '∗' can also be used. In addition, an IAXIS value of zero may be specified to indicate that the unmapping operation is to be applied to all the NDF's axes. Thus,

```
      CALL NDF_AUNMP( INDF, '*', 1, STATUS )
```

would unmap all the *axis* arrays for *axis* 1 of an NDF, while:

```
      CALL NDF_AUNMP( INDF, 'Centre', 0, STATUS )
```

would unmap the *centre* array for all of the axes.

19.9 Writing and Modifying Axis Arrays

New values may be written to an *axis* array by mapping it using NDF_AMAP and specifying a mapping mode of 'UPDATE' or 'WRITE'. Any new values assigned to the mapped array (or modifications made in the case of 'UPDATE' access) will then be written back to the NDF when the array is unmapped, as described in detail in §8.5. The following example shows how an *axis width* array might be mapped for write access, passed to a routine SETVAL which assigns values to it, and then unmapped:

```
      INTEGER PNTR( 1 ), EL

      ...

      CALL NDF_AMAP( INDF, 'Width', IAXIS, '_REAL', 'WRITE', PNTR, EL, STATUS )
      CALL SETVAL( 3.5, EL, %VAL( PNTR( 1 ) ), STATUS )
      CALL NDF_AUNMP( INDF, 'Width', IAXIS, STATUS )

      ...

*  Routine to assign axis width values.
      SUBROUTINE SETVAL( VALUE, EL, WIDTH, STATUS )
      INCLUDE 'SAE_PAR'
      INTEGER EL, STATUS
      REAL VALUE, WIDTH( EL )

      IF ( STATUS .NE. SAI__OK ) RETURN

      DO 1 I = 1, EL
         WIDTH( I ) = VALUE
 1    CONTINUE
      END
```

When using update access, NDF_AMAP will ensure that the mapped array is filled with the appropriate default values if the *axis* array is initially in an undefined state.

After successfully unmapping an *axis* array mapped for update or write access, the array's state will become defined. This process of assigning values to an *axis* array will also cause the NDF's *axis* coordinate system to become defined (see §19.3), so default values will be assigned to all

the NDF's *axis centre* arrays if these have not already been defined. This process takes place whenever NDF_AMAP is called with a mapping mode of 'UPDATE' or 'WRITE', and is exactly equivalent to calling the routine NDF_ACRE (§19.4) immediately beforehand.

19.10 Accessing Axis Variance Values as Standard Deviations

The values of an *axis variance* array may also be accessed directly as standard deviation values by specifying the special component name 'Error' in a call to NDF_AMAP. This facility operates in exactly the same way as the equivalent operation on the main *variance* component of an NDF (see §8.9) and causes the square root of the mapped values to be taken before they are returned. If a mapping mode of 'UPDATE' or 'WRITE' is specified, then the new or modified values are also squared before being written back to the *axis* array when it is unmapped.

Note that the component name 'Error' may only be used with mapping routines. The name 'Variance' should be used when later unmapping values accessed in this way.

19.11 Axis Normalisation Flags

The value of an *axis* normalisation flag (§18.8) may be obtained by means of the routine NDF_ANORM, as follows:

```
LOGICAL NORM

...

CALL NDF_ANORM( INDF, IAXIS, NORM, STATUS )
```

This will return the normalisation flag value for the specified *axis* via the logical NORM argument. An IAXIS value of zero may also be given, in which case the routine will return the logical "OR" of the results for each NDF *axis*. By default, the value returned will be .FALSE., indicating that no corrections to preserve *data* normalisation need be applied.

A new value for an *axis* normalisation flag may be set by using the NDF_ASNRM routine, as follows:

```
CALL NDF_ASNRM( NORM, INDF, IAXIS, STATUS )
```

The new flag value is supplied via the NORM argument. A value of zero for the IAXIS argument will cause the same normalisation flag value to be set for all the NDF's *axes*.

Note that the *axis* normalisation flag is regarded as an *axis* attribute (like numeric type and storage form) rather than an *axis* component, so setting a new normalisation flag value does not automatically cause the *axis* coordinate system to become defined. Normalisation flag values will only be retained if the *axis* coordinate system is in a defined state when the NDF is finally released from the NDF system (*i.e.* when the last identifier which refers to it is annulled).

19.12 The Numeric Type of Axis Arrays

An NDF's *axis* array values may be stored using any of the seven non-complex numeric types described in §7.1, and may also be accessed using any of these types. Type conversion will be performed automatically when required. By default, all *axis* arrays use a numeric type of '_REAL', although this may be changed if required (see below).

The routine NDF_ATYPE is provided for determining the numeric type of an *axis* array, as follows:

```
INCLUDE 'NDF_PAR'
CHARACTER * ( NDF__SZTYP ) TYPE

...

CALL NDF_ATYPE( INDF, 'Centre', IAXIS, TYPE, STATUS )
```

This will return the numeric type as an upper-case character string (*e.g.* '_REAL') via the TYPE argument. Note the use of the symbolic constant NDF__SZTYP (as defined in the include file NDF_PAR) to define the length of the character variable which is to receive the returned value.

NDF_ATYPE will also accept a list of *axis* array names, and a value of zero may be given for the IAXIS argument to indicate that all the NDF's axes should be considered at once. In this case, the routine will return the lowest-precision numeric type to which all the specified *axis* arrays may be converted without unnecessary loss of information.

So long as suitable access is available (see §23.1), the numeric type of an *axis* array may be changed at any time. If the array is in a defined state when this occurs, its values will be converted to the new type and will not be lost—the array may be reset beforehand (*e.g.* using NDF_AREST) if its values are not to be retained. A component list and/or an IAXIS value of zero may also be used. Thus, the following call would ensure that all of an NDF's *axis centre* and *width* values were stored in double-precision arrays:

```
CALL NDF_ASTYP( '_DOUBLE', INDF, 'Cen,Wid', 0, STATUS )
```

Note that the numeric type attribute of an *axis* array exists regardless of the array's state. A call such as that above can therefore be made before any *axis* values are assigned and will ensure that arrays of the required type are used when values are later assigned to them.

19.13 The Storage Form of Axis Arrays

An NDF's *axis* arrays may be stored using either *simple* or *primitive* storage form (see §12), the default being chosen to match that of the NDF's main *data* component. The storage form of an *axis* array may be obtained by using the routine NDF_AFORM, as follows:

```
INCLUDE 'NDF_PAR'
CHARACTER * ( NDF__SZFRM ) FORM

...

CALL NDF_AFORM( INDF, 'Centre', IAXIS, FORM, STATUS )
```

This example would return the storage form of the specified *axis centre* array (*e.g.* 'SIMPLE') as an upper-case character string via the FORM argument. Note the use of the symbolic constant NDF__SZFRM (defined in the include file NDF_PAR) to define the size of the character variable which is to receive the returned storage form information.

As with other NDF array components, the *primitive* storage form places certain restrictions on the use to which an *axis* array may be put (see §12.6). In practice, however, there is usually little need to be aware of this, because the NDF_ system will implicitly convert the storage form of *primitive axis* arrays to become *simple* whenever one of these restrictions would be violated. This is a straightforward operation which costs little. Since *bad*-pixel flags are not relevant to *axis* arrays, the only changes which can precipitate such a storage form conversion are those which modify the NDF's pixel-index bounds. The lower bound of an *axis* array is conceptually equal to the lower bound of the corresponding NDF dimension, so a *primitive axis* array will be converted to *simple* storage form if the lower bound of the relevant NDF dimension ceases to be equal to 1.

19.14 Accessing Axis Components via NDF Sections

The values of *axis* components and their attributes may be obtained (*i.e.* read) freely via identifiers which refer to NDF sections (see §15). In fact, in the case of *axis* character components there is no difference between using NDF sections and base NDFs for this purpose. With *axis* arrays, however, it is necessary that the appropriate part of each array be selected so that it correctly matches the NDF pixels to which the section refers. This operation is performed automatically by the NDF_ system.

In contrast, the writing or modification of *axis* component values and attributes has to be handled very differently when NDF sections are involved, in order to adhere to the principles described in §15.7. Accordingly, the following major restriction is placed on such operations:

No changes to axis component values or attributes may be made via an NDF section

This restriction is necessary so that an application which is applied to an NDF section is prevented from modifying axis values which may affect the interpretation of NDF pixels lying outside the section in question.

Applications should, nevertheless, still be able to operate on NDF sections with the possible limitation that *axis* modifications may be lost. To allow this, the NDF_ system permits attempts to modify axis values or attributes via NDF sections to proceed without error. However, all the relevant routines will simply return without action under these circumstances, so the attempted changes to the *axis* components will be disregarded, leaving the components unaffected.

19.15 Axis Extrapolation

When using NDF sections, the possibility also exists of reading *axis* array values from sections which refer to super-sets of the associated base NDF (§15.4). Although the section's *axis* arrays may be in a defined state (values having previously been assigned via the base NDF), parts of them will still lie outside the bounds of the base NDF, and so cannot have any values. In such cases, *axis* arrays must be *extrapolated* in order to return useful values. This process uses a set of *axis extrapolation rules* which are an extension of the rules normally used to generate the default *axis* values (§18.5), as follows:

Centre: *Axis centre* arrays are linearly extrapolated from the *centre* values of the two nearest base-NDF pixels on the same axis. If only one of these pixels exists, then the extrapolation assumes unit spacing between pixel *centres*.

Width: If an *axis width* array is in a defined state, it is extrapolated by duplicating the nearest base-NDF *width* value on the same *axis*. If it is in an undefined state, its new values are derived from the corresponding extrapolated *centre* values by employing the usual method for generating default *axis width* values (§18.5)—this normally gives rise to *width* values which match the pixel *centre* spacing of the two nearest base-NDF pixels on the same axis.

Variance: *Axis variance* arrays are always extrapolated by padding with the value zero.

There is also the possibility that an NDF section may have more dimensions than its associated base NDF (see §15.6), in which case associated *axis* components will also exist. These "virtual" *axis* components may be accessed in the same way as those associated with other dimensions, but they are always regarded as being in an undefined state. Default values will be supplied for them, if necessary, using the normal defaulting rules (see §18.5), but any new values assigned will simply be discarded.

20 CONNECTING WITH THE DATA SYSTEM

20.1 NDF Names

So far, examples of obtaining access to NDF data structures have depended on routines which use *parameters* and therefore require the support of a programming environment. However, it is also possible to use NDF_ routines in "standalone" applications and hence to access NDF structures whose location within the underlying data system (HDS) is specified explicitly, *i.e.* by name.

The name of an NDF dataset normally consists of up to three parts, as follows:[20]

```
<file_name><hds_path><subscripts>
```

Here, <file_name> is the name of the HDS "container file" which holds the data, <hds_path> is the HDS "path" which identifies the location of the HDS object (*i.e.* the NDF) within the container file, and <subscripts> is a parenthesised set of subscripts which may be used to select a *section* from the NDF (see §16). The container file name is always required, but the other two components are optional. Thus, a simple NDF name might look like:

```
mydatafile
```

(if the NDF were the top-level object in the container file, which is often the case), while a fairly complicated NDF name might resemble:

```
/users/bill/datafiles/today.RUN(66).BAND_B(1:66,1.04)
```

[20] Different rules may apply when accessing foreign format datasets and these are described in SSN/20.

Here, the first '.' separates the HDS path from the file name. Names of this sort may be supplied by users of NDF_ applications in response to prompts issued via a programming environment.

However, when NDF names are specified explicitly within applications, an alternative naming possibility exists because HDS files contain hierarchies of data structures rather similar to the directory structure of computer filing systems (see SUN/92 for a description of HDS data structures). Therefore, just as you can specify a file using an *absolute* name or a name *relative* to the current directory, so you can also specify the location of an NDF data structure within an HDS container file in an absolute or a relative fashion.

To give an *absolute* NDF name, you would specify it in full, as in the examples above,[21] while to give a *relative* NDF name, you would supply an active HDS locator and the name of the NDF data structure *relative* to the HDS object that the locator identifies.

To allow this, most NDF_ routines that handle the names of NDFs will accept two arguments: an HDS locator and an associated character string containing a name. The following section illustrates how these are used.

20.2 Finding and Importing NDFs

Using the example above, suppose that LOC is an active HDS locator associated with the data structure:

```
/users/bill/datafiles/today.RUN(66)
```

The routine NDF_FIND could then be used to find an NDF data structure relative to this object, and to import it into (*i.e.* make it known to) the NDF_ library, as follows:

```
CALL NDF_FIND( LOC, 'BAND_B(1:66,1.04)', INDF, STATUS )
```

(note that here we have specified a *relative* name for the second argument). This would find the NDF section whose absolute name is:

```
/users/bill/datafiles/today.RUN(66).BAND_B(1:66,1.04)
```

An identifier for this NDF would be returned via the INDF argument and the data structure could then be manipulated using other NDF_ routines. The locator passed to NDF_FIND may be annulled afterwards without affecting the subsequent behaviour of the NDF_ system.

Note that the relative name supplied could be any trailing fragment of the full (absolute) name, and might consist simply of a subscript – so long as a suitable locator was also available. In fact, the relative name could be entirely blank, in which case the locator supplied would be taken to identify the NDF directly. Thus, in:

```
CALL NDF_FIND( LOC, ' ', INDF, STATUS )
```

[21] Note that an *absolute* NDF name may nevertheless still start with a container file specified using a *relative* file name.

the locator LOC should be associated with the NDF data structure itself.

To specify an NDF in a call to NDF_FIND using an *absolute* name instead, we utilise the special locator value DAT__ROOT, which is provided by HDS and defined in the include file DAT_PAR. This locator represents the notional HDS object which is the parent of all other HDS objects,[22] and specifying it as a locator value allows the associated NDF name to be an *absolute* name. Thus:

```
      INCLUDE 'DAT_PAR'

      ...

      CALL NDF_FIND( DAT__ROOT,
     :               '/users/bill/datafiles/today.RUN(66).BAND_B(1:66,1.04)',
     :               INDF, STATUS )
```

would access the same NDF section as in the examples above, this time using its *absolute* name.

20.3 A Note on Modes of Access

When obtaining an NDF identifier for a data structure using NDF_FIND, the NDF library has to decide what mode of access to grant (*i.e.* whether it should allow the NDF to be modified or whether it should be "read only"). It normally does this by inspecting the locator supplied and permitting modification of the NDF only if the locator itself permitted modification of the data structure. Access can subsequently be further restricted, if required, using NDF_NOACC (see §23.1).

However, if the locator supplied has the value DAT__ROOT (indicating that an absolute NDF name has been given), then this method cannot be applied. In this case, read only access will be obtained by NDF_FIND and subsequent modification of the NDF will not be permitted. If the ability to modify the NDF is required, the routine NDF_OPEN should be used instead, as this allows the access mode to be specified (see §20.10).

20.4 Obtaining an HDS Locator for an NDF

The NDF_ system also allows you to obtain an HDS locator for a data structure whose NDF identifier you supply. In effect, this is the reverse of the importation process described in §20.2. It is performed by the NDF_LOC routine, as follows:

```
      CALL NDF_LOC( INDF, MODE, LOC, STATUS )
```

The MODE argument specifies the mode of access required ('READ', 'UPDATE' or 'WRITE') and LOC returns the resulting HDS locator. Note that you should annul this locator (*e.g.* using DAT_ANNUL) when it is no longer required, as the NDF_ system will not perform this task itself.

[22] In essence it stands for the filing system of the host computer.

20.5 NDF Placeholders

Routines are also provided for the creation of NDFs at specified locations within the data system, rather than indirectly via parameters. These routines depend on the concept of a *placeholder* for their operation.

In many ways, a placeholder is similar to an NDF identifier; *i.e.* it is an integer value which the NDF_ system issues to identify an entity about which it holds information. In this case, however, the entity is not a data structure, but a position within the data system at which an NDF will be created at some later time.

An NDF placeholder may be obtained by calling the routine NDF_PLACE and specifying a name for the new NDF using an HDS locator and an absolute or relative name in the same way as when accessing an existing NDF (§20.2). Thus, if LOC is a locator associated with an existing HDS structure, then:

```
CALL NDF_PLACE( LOC, 'NEW_NDF', PLACE, STATUS )
```

would return an integer value PLACE, which is a placeholder for a new NDF called 'NEW_NDF' contained within that structure. Similarly, the absolute name of a new NDF could be specified as follows:

```
INCLUDE 'DAT_PAR'

...

CALL NDF_PLACE( DAT__ROOT, 'file.STRUCT(2,2).NEW_NDF', PLACE, STATUS )
```

All the HDS structures residing at levels above the actual NDF object to be created must already exist, otherwise NDF_PLACE will fail.

Sometimes, it is more convenient if an object is already in existence at the location in the data system for which a placeholder is to be issued. For instance, this allows placeholders to refer to the individual elements of an array of structures which has previously been created, so that arrays of NDFs may be built. Any pre-existing object for which an NDF placeholder is to be issued must be a scalar structure and must have an HDS type of 'NDF'. It must also be empty (*i.e.* it must have no components). NDF_PLACE is then called in the usual way. In the following example, for instance, a 1-dimensional array of structures is created with 5 elements, and a placeholder is then obtained for the second element:

```
INTEGER DIM( 1 )

...

DIM( 1 ) = 5
CALL DAT_NEW( LOC, 'NDF_ARRAY', 'NDF', 1, DIM, STATUS )
CALL NDF_PLACE( LOC, 'NDF_ARRAY(2)', PLACE, STATUS )
```

As with NDF_FIND, a blank name string may be supplied to NDF_PLACE to indicate that the data system location to be used is identified directly by the locator; this mode of use is only applicable when an object already exists at that location.

20.6 Creating NDFs via Placeholders

The only valid thing that can be done with an NDF placeholder is to pass it to a routine which creates an NDF. Thus, to create a new simple NDF by this means, the routine NDF_NEW might be used along with NDF_PLACE, as follows:

```
CALL NDF_PLACE( LOC, 'NEW_NDF', PLACE, STATUS )
CALL NDF_NEW( '_INTEGER', NDIM, LBND, UBND, PLACE, INDF, STATUS )
```

The call to NDF_NEW will create the required new simple NDF (in this case with a type of '_INTEGER') in place of the placeholder PLACE; *i.e.* in component NEW_NDF of the HDS structure with locator LOC. This act of creation automatically *annuls* the placeholder, whose value is reset to NDF__NOPL (this value is defined in the include file NDF_PAR and is reserved for indicating that a variable is not a valid placeholder).

A similar routine NDF_NEWP also exists by analogy with NDF_CREP (see §13.4) to create a primitive NDF via a placeholder in exactly the same way.

Note that placeholders are only intended for local use within an application and only a limited number of them are available simultaneously. They are always annulled as soon as they are passed to a routine which creates an NDF, where they are effectively exchanged for an NDF identifier. Since this is the only valid fate for a placeholder, there is no routine for annulling them explicitly. However, the routine NDF_END will annul any which are left outstanding as part of the "cleaning up" role it performs at the end of an NDF context (see §3.4).

20.7 Temporary NDFs

Placeholders also provide a convenient means of obtaining "scratch" NDFs for temporary use within an application. In this case, exactly the same NDF creation routine can be used, but the placeholder is obtained using the routine NDF_TEMP, which returns a placeholder for a *temporary* NDF. To create a temporary simple NDF, the following calls might be used:

```
CALL NDF_TEMP( PLACE, STATUS )
CALL NDF_NEW( '_INTEGER', NDIM, LBND, UBND, PLACE, INDF, STATUS )
```

The resulting NDF identifier INDF may be used in exactly the same way as an identifier for a permanent NDF, except that the data structure it is associated with will not be retained after it has been finished with. More specifically, a temporary NDF will be deleted as soon as the last NDF identifier associated with it is annulled, either explicitly (*i.e.* with NDF_ANNUL) or implicitly (*e.g.* as part of the cleaning up performed by NDF_END).

It is possible to determine whether an existing NDF identifier refers to a temporary NDF using the routine NDF_ISTMP:

```
CALL NDF_ISTMP( INDF, ISTMP, STATUS )
```

A logical value of .TRUE. is returned via the ISTMP argument if the NDF is temporary.

20.8 Copying NDFs

Placeholders also play a role in the copying of NDF data structures from one location to another within the data system. In this case they are used to identify the destination for the copying operation, which is performed by the routine NDF_COPY. This should be preceded by a call which generates a suitable placeholder. For instance:

```
CALL NDF_PLACE( LOC, 'MY_NDF_COPY', PLACE, STATUS )
CALL NDF_COPY( INDF1, PLACE, INDF2, STATUS )
```

Here, the placeholder causes NDF_COPY to copy the NDF with identifier INDF1 to form a new NDF in the component MY_NDF_COPY of the HDS structure with locator LOC. As usual, the placeholder is annulled by the copying operation and a new NDF identifier INDF2 is issued to refer to the new data structure. This new structure contains all the information which was present in the original. It is possible to copy both base NDFs and NDF sections in this way, but the copying operation always creates a new base NDF.

Naturally, it is also possible to create a temporary copy of an NDF by using NDF_TEMP to obtain the placeholder, as follows:

```
CALL NDF_TEMP( PLACE, STATUS )
CALL NDF_COPY( INDF1, PLACE, INDF2, STATUS )
```

20.9 Selective Copying of NDF Components

Copying of selected NDF components is also possible by using the routine NDF_SCOPY. This behaves in the same way as NDF_COPY, except that it also accepts a list of the NDF components to be copied as its second argument, thus:

```
      CALL NDF_SCOPY( INDF1, 'Data,Var,Axis,Nohist,Noext(CCDPACK)',
     :                PLACE, INDF2, STATUS )
```

The component list has the same syntax, and is used in exactly the same way, as in the routine NDF_PROP (see §§14.5–14.8). In fact, NDF_SCOPY is the "standalone" equivalent of NDF_PROP and similar rules about the propagation of NDF component information should be observed when using it (see §14).

20.10 General Access to NDFs

The routine NDF_OPEN is a general-purpose routine for accessing NDF data structures. It will perform the same tasks as NDF_FIND and NDF_PLACE but, although it is a little more complicated, it also offers some additional features. In particular, using NDF_OPEN to access an NDF is similar to using a Fortran OPEN statement to access a file, in that it allows you to specify whether you think the NDF already exists or not. It also allows you to specify an access mode.

If the fourth (STAT) argument to NDF_OPEN is set to 'OLD', the routine behaves like NDF_FIND, allowing you to access an existing NDF. For example:

```
CALL NDF_OPEN( DAT__ROOT, 'any_ndf', 'UPDATE', 'OLD', INDF, PLACE, STATUS )
```

would obtain 'UPDATE' access to the specified NDF and return an identifier for it via the INDF argument (in this case the PLACE argument is not used and returns the value NDF__NOPL). An error would result if the NDF did not exist.

If the fourth argument is set to 'NEW', NDF_OPEN instead behaves like NDF_PLACE. A placeholder for the specified NDF is returned via the PLACE argument, and the INDF argument (which is not used in this case) returns the value NDF__NOID.

Finally, if its fourth argument is set to 'UNKNOWN', then NDF_OPEN first attempts to access an existing NDF data structure using the locator and name supplied. If it succeeds, an identifier for the NDF is returned via the INDF argument. However, if it fails because the NDF does not exist, it instead creates a new placeholder for it and returns this via the PLACE argument. You can tell which of these two possibilities occurred by examining the INDF and PLACE arguments on return – only one of these will differ from its "null" value (NDF__NOID or NDF__NOPL respectively).

A typical use of NDF_OPEN might be to access an existing NDF if it exists, or to create a new one if it does not, as follows:

```
      CALL NDF_OPEN( LOC, 'CALIB.BAND_B', 'UPDATE', 'UNKNOWN',
     :               INDF, PLACE,  STATUS )
      IF ( PLACE .NE. NDF__NOPL ) THEN
         CALL NDF_NEWP( '_INTEGER', 1, UBND, PLACE, INDF, STATUS )
      END IF
```

20.11 Deleting NDFs

Deletion of an NDF may be performed by calling the routine NDF_DELET, thus:

```
      CALL NDF_DELET( INDF, STATUS )
```

If the necessary access is available (see §23.1), the associated data structure will be erased and a value of NDF__NOID will be returned for the INDF argument. Any other identifiers associated with the same data structure will also become invalid at this point. Note that deletion cannot be performed via an identifier which refers to an NDF section. In this case, NDF_DELET will simply annul the identifier as if NDF_ANNUL had been called (see §3.3).

21 ALTERING BOUNDS AND PIXEL INDICES

21.1 Setting New Pixel-Index Bounds and Dimensionality

The pixel-index bounds of an NDF may be altered when required by explicitly setting them to new values using the routine NDF_SBND. The dimensionality of the NDF may also be changed at the same time. For instance:

```
      INTEGER NDIM, LBND( NDIM ), UBND( NDIM )
```

```
...

CALL NDF_SBND( NDIM, LBND, UBND, INDF, STATUS )
```

will change an NDF's dimensionality to the value specified by the NDIM argument, and set its lower and upper pixel-index bounds to the values specified in the LBND and UBND arrays. When an NDF's shape is changed in this way, the pixel values of any array component which is in a defined state may be affected, as follows:

Retained Pixels – These are pixels with indices which lie within both the initial and final NDF pixel-index bounds. The values of these pixels are retained.

New Pixels – These are pixels with indices which lie within the new pixel-index bounds, but did not exist within the initial bounds. These pixels will be assigned the appropriate *bad*-pixel value (see §9). If such pixels are introduced, then the *bad*-pixel flag of affected NDF components will be updated to reflect this fact.

Lost Pixels – These are pixels with indices which lie within the initial NDF pixel-index bounds, but outside the new bounds. The values of such pixels are lost and cannot be recovered, even by changing the NDF's bounds back to their original values.

In cases where the dimensionality of the NDF also changes, the association between pixel indices before and after the change is established in the same way as when creating NDF sections (see §15.6), *i.e.* by padding the bounds with 1's to match the dimensionalities.

Note that altering the shape of an NDF section using NDF_SBND is a relatively inexpensive operation which merely involves changing the shape of the "window" into the NDF which the section describes. In contrast, altering the shape of a base NDF causes changes to be made to the actual data structure and can take considerably longer, especially if one or more of the NDF's array components are in a defined state and contain values which may need to be shuffled to accommodate the change. Consequently, if the values of any array components need not be retained, they should be reset to an undefined state before changing the shape of a base NDF. For instance:

```
CALL NDF_RESET( INDF, 'Data,Variance,Quality', STATUS )
CALL NDF_SBND( NDIM, LBND, UBND, INDF, STATUS )
```

would ensure that no array values are retained and that no time is unnecessarily wasted as a result.

21.2 Applying Pixel-Index Shifts

An alternative way of changing the pixel-index bounds of an NDF is to apply shifts to its pixel indices using the routine NDF_SHIFT. For instance:

```
INTEGER NSHIFT, SHIFT( NSHIFT )

...

CALL NDF_SHIFT( NSHIFT, SHIFT, INDF, STATUS )
```

would apply a set of NSHIFT shifts to an NDF (one to each dimension) as specified in the integer array SHIFT. As a result, the pixel-index bounds and the indices of each pixel in the NDF would be changed by the amount of shift applied to the corresponding dimension. The shifts applied may be positive or negative. Thus, if the set of shifts (10,1,−3) were applied to an NDF with shape:

(10:20, 7, 0:5)

then its shape would change to become:

(20:30, 2:8, −3:2)

Note that the behaviour of NDF_SHIFT and NDF_SBND is quite different. With NDF_SBND (§21.1) the pixel indices remain fixed while the NDF bounds move, so that pixels can be lost from the edges of the NDF and new ones can be introduced. With NDF_SHIFT, however, the pixel indices move with the bounds, so that no pixels can ever be lost and no new ones are introduced. NDF_SHIFT also preserves the dimension sizes of the NDF.

The application of pixel-index shifts with NDF_SHIFT is a relatively inexpensive operation. When applied to an NDF section, the change in pixel indices applies only to that section (and any identifiers subsequently derived from it) and causes no permanent change to the base NDF or to other sections. When applied to a base NDF, however, the actual data structure is altered and this will be apparent through any other base NDF identifiers which refer to it. Note, however, that sections previously derived from a base NDF are not affected if NDF_SHIFT is applied to the base NDF (*i.e.* such sections will retain their original pixel indices and values).

22 THE HISTORY COMPONENT

22.1 Purpose of the History Component

The *history* component of an NDF provides a way of recording the processing operations which are performed on the dataset. It consists of a series of *history records* (with some ancillary information) each of which contains information about a particular processing operation and when it occurred. Normally, each of these operations will correspond with the execution of a single application which has either modified or created the NDF.

It is intended that the user of NDF applications should have considerable control over the recording of history information, but that writers of applications should not generally need to concern themselves with how this is achieved. Indeed, if a user has access to basic utilities for manipulating the *history* component of NDFs, then new applications will automatically provide history recording facilities without their authors making any specific provision for it. Most programmers may therefore need to read very little of this section except in special cases where explicit control over history recording is required.

22.2 The History Component's State

As with other NDF components, the *history* component has a *state*, which is represented by one of the two values .TRUE. or .FALSE.. If the state is .FALSE., the *history* component is not defined, and the NDF contains no history information and no reference can be made to its *history* component. However, if the state is .TRUE., then the *history* component is defined and enquiries can be made about it. In this case, *history* records may also be present which describe the past processing history of the NDF. If it is a new NDF, however, it is possible that no such records may yet have been written to the *history* component.

The state of an NDF's *history* component can be determined using the NDF_STATE routine by specifying a component name of 'History', thus:

```
CALL NDF_STATE( INDF, 'History', STATE, STATUS )
```

The state information is returned via the LOGICAL argument STATE.

22.3 Preparing to Record History Information

Unlike other NDF components, the *history* component does not become defined simply by writing to it (*i.e.* by the act of recording new history information). This is because recording of history information is not always required. It can, for instance, lead to the use of large amounts of file space or of additional processing time which is not justified, so resetting the *history* component (setting its state to .FALSE. – see §22.4) provides the user of an application with one way of disabling history recording when it is not required. In general, one does not want this action negated by the next application which attempts to record further history information.

Before history information can be recorded in an NDF, it is therefore necessary to explicitly define a *history* component to receive it. The NDF_HCRE routine will perform this, as follows:

```
CALL NDF_HCRE( INDF, STATUS )
```

NDF_HCRE will return without error if a *history* component is already defined. Otherwise, it will initialise a new one, making the NDF receptive to new *history* records (although there will initially be none of these present).

In addition, if the "NDF_AUTO_HISTORY" environment variable or "AUTO_HISTORY" tuning parameter is set to a non-zero integer (see NDF_TUNE), then a History component will be added automatically to NDFs created using either NDF_CREAT or NDF_NEW.

22.4 Resetting the History Component

The opposite process, of resetting the *history* component to an undefined state, may be performed using the NDF_RESET routine with a component name of 'History', thus:

```
CALL NDF_RESET( INDF, 'History', STATUS )
```

This has the effect of rendering the *history* component undefined and of **deleting all the information it contains**. Because this operation is irreversible and destroys all previous history

information, it is performed only if explicitly requested. In particular, the *history* component remains unaffected when a "wild-card" component name of '*' is given to NDF_RESET, or when an existing NDF is opened for 'WRITE' access (circumstances in which other NDF components are normally automatically reset).

A less drastic method of disabling history recording while retaining earlier information is controlled by the *history update mode* which is described in §22.8.

22.5 Default History Recording

As its name suggests, *default history recording* provides a means of recording information in the *history* component of an NDF in cases where the programmer has not taken any explicit action to generate this information within an application. This is expected to be the norm.

For *default history recording* to occur, the NDF's *history* component must be in a defined state (as explained above). Control over this is normally in the hands of the user of the application, who can run a separate utility to modify the *history* component of specified NDFs and hence nominate which of them are to accumulate history information.

In addition, the NDF must have been opened either for 'WRITE' or 'UPDATE' access, so that the current application will have modified (or created) it. Some further conditions are also necessary, as discussed later, but no explicit action is required on the part of the programmer when writing the application. Thus, *default history recording* is available to even the most rudimentary of NDF applications.

Default history information is normally written to the *history* component of an NDF when it is released from the NDF_ system, either by a call to NDF_ANNUL which annuls the last valid identifier for the NDF, or implicitly via the cleaning-up action of the NDF_END routine (see §3.4). In any event, the information written will be the same, consisting of the following items:

- • APPLICATION – Name of the application which wrote the information
- • DATE – Date and time the information was written
- ○ USER – User ID for the person who ran the application
- ○ HOST – Name of the machine on which the application was running
- ○ REFERENCE – Reference name, which fully identifies the NDF dataset
- • TEXT – Free-format text containing any additional information

The first five of these items contain "fixed" information which is always automatically associated with any new *history* record and which may be read back at a later date and acted upon.[23]

The final TEXT item is "free format" and there is no restriction on what this may contain. It is available for applications to write their own history information, either to augment or to replace that written by default. When written by default, this *history* text will typically record the

[23]Except that the USER, HOST and REFERENCE items (marked by open circles) may not always be present in *history* records written by applications which do not use the NDF_ library. In this case, enquiries about them will simply result in a blank value being returned. Also, DATE can optionally be set to an alternative, user-supplied value using the NDF_HSDAT routine.

command line used to invoke the application (or, depending on the programming environment in use, the values of its parameters) together with the name of the file which was executed. The precise content and format of the default history text may depend on a number of factors,[24] so you should not write software which depends upon such details.

22.6 Propagation of History Information

Not all NDF data structures are created from scratch and then successively modified *in situ*. More commonly, they are produced by copying a certain amount of information from related input datasets while applying the required modifications in the process. In this situation, one of the input datasets is normally regarded as *primary*, and it is from this that the output NDF inherits most of its ancillary information by the process termed *propagation* (see §14).

To be complete, a description of the processing history of an NDF generated in this way would need to contain the processing histories of all the input datasets which have contributed to it. However, it is generally regarded as unnecessary to retain this wealth of history information as it would lead to exponential growth in the amount of information to be processed and stored. A more practical proposition, and the one supported by the NDF_ library, is to *propagate* the past history from **only** the *primary* input dataset to the output, and then to update this by appending a new record to reflect the action of the current application.

As a consequence, a complete record of all previous events will not be present in the new NDF. However, if the history information recorded by each application includes the full names of its input datasets, then these can be inspected separately to recover any further information. This keeps the amount of history information within reasonable bounds. Because the NDF_ library stores the name of the NDF in which a *history* component resides whenever it creates a new *history* record,[25] an audit trail is automatically produced which allows the "ancestors" of any dataset (*i.e.* those datasets from which the history component has previously been propagated) to be identified.

Propagation of *history* component information takes place in a similar way to the propagation of any other NDF component (see §14) and is typically performed by the NDF_PROP (or NDF_SCOPY) routine. In this context, the *history* component is considered "safe" in the sense that its validity is not affected by the processing performed by most applications. It is therefore propagated by default, and you must explicitly specify if you do not want it to be propagated. This means that in practice most applications need take no action to ensure that history information is kept, since it will be maintained automatically via the *propagation* and *default history recording* mechanisms.

Some applications create new NDFs from scratch using (for instance) NDF_NEW or NDF_CREAT, rather than by propagation via NDF_PROP. In many cases, such applications will still want to propagate history information to the new NDF from some specified input NDF. This can be achieved using NDF_HCOPY.

22.7 Explicitly Controlling History Text

When writing an application, you have the option of accepting the *default history recording* provided by the NDF_ library, or you may want to add further information of which the NDF_

[24] Such as the software environment or operating system in use, the version of the NDF_ library, and the *history update mode* – see §22.8.

[25] This information is stored in the REFERENCE item – see §22.5.

system is unaware (a record of the results of a calculation, for instance). Not only is it possible to add such information, either prefixing or appending it to the default *history* text, but it is also possible to replace the default *history* text entirely if required.

The routine which allows this is NDF_HPUT, which writes new textual information to a *history* record, thus:

```
CHARACTER HMODE * ( NDF__SZHUM ), TEXT( 2 ) * ( NDF__SZHIS )
LOGICAL REPL, TRANS, WRAP, RJUST

...

HMODE = 'NORMAL'
APPN = ' '
REPL = .FALSE.
TEXT( 1 ) = 'This is explicit history information (line 1)...'
TEXT( 2 ) = '   ...and this is the second line.'
TRANS = .FALSE.
WRAP = .FALSE.
RJUST = .FALSE.
CALL NDF_HPUT( HMODE, APPN, REPL, 2, TEXT, TRANS, WRAP, RJUST, INDF,
               STATUS )
```

In this example, NDF_HPUT is being used to append two lines of text, stored in the TEXT array, to the text of the current *history* record for an NDF – remember, a new *history* record is normally created for each application which executes. (The HMODE, APPN, TRANS, WRAP and RJUST arguments can be ignored for the moment; their use is described in subsequent sections.)

If this is the first history information to be written to this particular NDF by the current application, then there will not yet be a *history* record to contain it. In this case, NDF_HPUT will create a new record and initialise it, recording the date and time and other standard information. The text provided will then form the first two lines of the text associated with the *history* record. If a current record already exists, NDF_HPUT will instead simply append the text to whatever may have been written earlier.

Normally, default history information is not written by the NDF_ system until an NDF is released, so information written by NDF_HPUT in the manner above will be prefixed to this default information. If a different order is required, then the routine NDF_HDEF may be used to force the default history information to be written prematurely, thus:

```
CALL NDF_HDEF( INDF, STATUS )
```

It will then not be written when the NDF is released. This makes it possible to interleave explicit and default *history* text in any order.

It is also possible to prevent default information from being written at all by setting the REPL argument of NDF_HPUT to .TRUE., indicating that the new text is to replace that provided by default. If the default information has not yet been written (normally the case), it will be completely suppressed by this action. A single such call to NDF_HPUT is sufficient and *default history recording* cannot then be re-established until the NDF has been released (or a new application started – see §22.18).

Each history record has a date and time associated with it. By default, this will be the UTC date and time at which the history record was created. However, an alternative date and time can be specified by calling the NDF_HSDAT routine before the history record is created.

22.8 The History Update Mode

As already indicated, there may be circumstances where finer control over the recording of history information is required than is provided simply by the state of the *history* component. For instance, you might want to disable further recording of history information for an NDF without losing the record of its past processing history. There may also be cases in which rather brief history information is required in order to save file space, or other circumstances where the fullest possible information is required (perhaps if the processed data were intended for archiving).

These requirements are met by the *history update mode* associated with an NDF's *history* component. It may have any of the following states, with the associated meanings:

'DISABLED'	No history recording is to take place
'QUIET'	Only brief history information is to be recorded
'NORMAL'	Normal history recording is required
'VERBOSE'	The fullest possible history information is required

When a *history* component is first defined, its update mode defaults to 'NORMAL'. This may subsequently be altered using the NDF_HSMOD routine, so long as 'WRITE' access is available for the NDF, as follows:

```
CALL NDF_HSMOD( 'DISABLED', INDF, STATUS )
```

The update mode string may be abbreviated (to not less than 3 characters).

The update mode can be retrieved from an NDF using NDF_HGMOD.

The *history update mode* is a permanent attribute of the *history* component and remains with it to affect subsequent recording of history information, although it may be given a new value at any time. It affects both *default history recording* and (potentially) history information written explicitly by applications, according to its value at the time the information is written.

As might be expected, the HMODE argument of NDF_HPUT (set to 'NORMAL' in the example in §22.7) corresponds with the *history update mode* argument of NDF_HSMOD. In this case, it specifies the "priority" which the history information should have and takes one of the following values:

'QUIET'	Highest priority
'NORMAL'	Normal priority
'VERBOSE'	Lowest priority

The text supplied to NDF_HPUT will actually be written to the NDF only if:

(1) The NDF's *history* component is defined, and

(2) Its *history update mode* is not 'DISABLED', and

(3) The priority specified in the call to NDF_HPUT is at least as great as that of the NDF's current *history update mode*.

Otherwise, NDF_HPUT will simply return without action (or error).

The purpose of this is to allow applications to tailor their explicit recording of history information to the setting of the *history update mode* attribute of the NDFs they are processing. Thus, most *history* text might be classed as 'NORMAL' but additional information classed as 'VERBOSE' could also be provided and would only be available to users who request it (by setting the *history update mode* of their NDFs appropriately). Really vital information should be classed as 'QUIET', and will then always be delivered unless history recording has been completely disabled.

If the HMODE argument of NDF_HPUT is left blank, it defaults to 'NORMAL'.

22.9 Using Message Tokens in History Text

The NDF_HPUT routine facilitates the inclusion of variable values in *history* records by allowing *message tokens* to be optionally embedded within the lines of text supplied to it. If this facility is required, then the TRANS argument should be set to .TRUE. to request message token translation, and the tokens used should be defined beforehand – for instance, by calling the appropriate MSG_ routines (see SUN/104 which includes a full description of message tokens). In the following example, the results of an earlier calculation are inserted into a *history* record in this way:

```
CALL MSG_SETI( 'NPIX', N )
CALL MSG_SETR( 'VALUE', RESULT )
CALL MSG_SETC( 'UNITS', CVAL )
TEXT( 1 ) = 'Result of summing ^NPIX pixels:'
TEXT( 2 ) = '                        ^VALUE (^UNITS)'
TRANS = .TRUE.
CALL NDF_HPUT( HMODE, APPN, REPL, 2, TEXT, TRANS, WRAP, RJUST, INDF,
               STATUS )
```

22.10 History Text Width

The text stored in *history* records may have any width (or line length) ranging from 1 to NDF__SZHMX[26] characters, but all the lines of text associated with a given record have the same width (*i.e.* the same upper limit on the number of characters they may contain). This width is determined when the first line of text is written to the record.

If the first line of text in a *history* record is written by the NDF_ system itself (when writing default history information, for instance), then the text width for that record will be initialised to NDF__SZHIS[27] characters. This is the recommended width for *history* text. However, if the NDF_HPUT routine writes the first line, then the text width will be determined by the length of each element of the TEXT array supplied to it, as returned by the Fortran intrinsic LEN function. An error will result if this is too long (*i.e.* exceeds NDF__SZHMX).

[26]The NDF__SZHMX constant is defined in the include file NDF_PAR and currently has the value 200.

[27]Also defined in the NDF_PAR include file, currently to be 72 characters.

22.11 Formatting Options for History Text

Because the text width associated with a *history* record is not fixed, there can sometimes be difficulty in knowing how to format *history* text correctly, especially when appending to an existing record. In principle, it would be possible to use the enquiry routine NDF_HINFO to determine the text width and to act accordingly, but in practice this is cumbersome and rarely worth the effort. In addition, if message tokens are present within the text, it becomes difficult to predict the length of each line after token substitution has taken place.

To address this problem, NDF_HPUT provides several formatting options aimed at reconciling the possibly different widths of the text supplied (after message token substitution if appropriate) and the *history* record into which it must be written. The guiding principle is that no *history* text should ever be discarded as a result of limited line length, and that as far as possible it should remain legible.

The simplest formatting option is selected by setting both the WRAP and RJUST arguments of NDF_HPUT to .FALSE., in which case input text lines will simply be broken if they exceed the width available and continued on a new line (line breaking will take place at a suitable space if possible). This option is normally most suitable for "fixed-format" output of results.

If its WRAP argument is set to .TRUE., NDF_HPUT will instead perform "paragraph wrapping" on the text supplied. In this case, the text will be re-formatted so that the maximum number of words appears on each line, with words separated by single spaces. Re-formatting will not occur across "paragraph boundaries" marked by blank input lines. This option is most suitable for prose-style continuous text, particularly if it contains elements of unknown length such as message tokens. It also has the advantage of minimising the space needed to store the text.

Finally, if the RJUST argument of NDF_HPUT is set to .TRUE., it will "right justify" the text wherever a line is broken for continuation on the next line (*i.e.* after first performing either of the formatting operations described earlier if necessary). This results in lines of the final text being padded with blanks so as to produce a uniform right margin occupying the full text width (normally the right margin would remain ragged). When wrapping text, the final line of each paragraph, which may naturally be shorter than other lines, is not padded, and neither are unbroken lines if wrapping is disabled. This option is largely cosmetic but may improve the readability of some types of text. It does not affect the amount of space used.

22.12 Automatic Error Recording

One event which it is usually very important to know about is the failure of an application to complete properly. This could be because something unrecoverable went wrong and the application decided to terminate prematurely, or it could simply be that the user requested that it should abort. In either case, it is important that any datasets modified by that application should contain some record that it did not complete normally, so that they can later be identified and their contents regarded with appropriate suspicion.

The NDF_ library provides this service automatically via its history recording mechanism. To do this, it inspects the value of the STATUS argument passed to any routine which can potentially cause NDF datasets to be released from the NDF_ system (NDF_ANNUL and NDF_END). If the STATUS value is not equal to SAI__OK,[28] this is taken to indicate that an error has occurred

[28] As defined in the include file SAE_PAR.

and that any NDFs released at that point may contain erroneous data. To record this fact, the value of STATUS and the text of any pending error messages (as previously reported through the ERR_ and EMS_ routines – see SUN/104 and SSN/4) will be appended to the current *history* record before each NDF is released. Rather similar action is also taken by the special routine NDF_HEND (although it does not itself cause NDF datasets to be released – see §22.18).

This error recording facility operates somewhat like NDF_HPUT with an effective priority of 'QUIET', so recording of errors will take place regardless of the current *history update mode* setting. However, it will not occur if history recording is completely disabled.

22.13 Enquiring about Past History Information

After several applications have processed an NDF and recorded history information, the NDF's *history* component will contain a series of records, each identified by a date and time, which document its processing history.

You can determine how many such records exist, either with the general history enquiry routine NDF_HINFO, or more directly using the NDF_HNREC routine, thus:

```
CALL NDF_HNREC( INDF, NREC, STATUS )
```

The number of records is returned via the INTEGER argument NREC. Note that this value may (or may not) include a *history* record for the current application, depending on whether any history information has yet been written by it. The value may also be zero if no history information has ever been written.

Information about a particular record may be obtained by calling NDF_HINFO and supplying the record number.[29] For example, if history recording has been in operation, you could obtain the name of the application which most recently modified an NDF as follows (the result is returned in the APPN variable):

```
INCLUDE 'NDF_PAR'
CHARACTER * ( NDF__SZAPP ) APPN

...

CALL NDF_HNREC( INDF, NREC, STATUS )
CALL NDF_HINFO( INDF, 'APPLICATION', NREC, APPN, STATUS )
```

Other information about individual records may also be obtained using NDF_HINFO, such as the text width, the number of text lines written and, of course, the date and time associated with the record (see Appendix D for full details of the information items available).

22.14 Accessing History by Date and Time

In addition to the record number, the date and time form a convenient means of addressing *history* records which are, necessarily, stored in chronological order. To facilitate this, the NDF_HFIND routine provides a search facility which allows records to be identified by date and time:

[29] History record numbers start at 1.

```
      INTEGER YMDHM( 5 )
      LOGICAL EQ
      REAL SEC

      ...

      EQ = .FALSE.
      CALL NDF_HFIND( INDF, YMDHM, SEC, EQ, IREC, STATUS )
```

Given a date and time (with the years, months, days, hours and minutes fields stored in the YMDHM array and the seconds field held in SEC), this routine will return the number of the first record written after the specified date and time. A record number of zero is returned if no such record exists.

In this example, the EQ argument is set .FALSE., which instructs the routine not to report an exact date/time match. By setting it to .TRUE., an exact match would be permitted. This makes it possible to select records according to any constraint on the date and time of creation, using combinations of the following techniques:

- Varying the value of the EQ argument,
- Selecting the record which precedes or follows that actually returned,
- Using several calls to NDF_HFIND in succession.

22.15 Accessing Past History Text

The *history* text is perhaps the item of most interest associated with a *history* record. No routine is provided for accessing this directly, but NDF_HOUT may be used to pass it to a service routine, which may then handle it in any way required. Most commonly, this will involve displaying the text, for which a default service routine NDF_HECHO is provided. NDF_HECHO is a very simple routine and is described in Appendix D, but an outline of how it works is given here:

```
      SUBROUTINE NDF_HECHO( NLINES, TEXT, STATUS )
      INCLUDE 'SAE_PAR'

      INTEGER NLINES, STATUS, I
      CHARACTER TEXT( NLINES ) * ( * )

*  Check status.
      IF ( STATUS .NE. SAI__OK ) RETURN

*  Print out each line of text.
      DO 1 I = 1, NLINES
         CALL MSG_FMTC( 'TEXT', '3X,A', TEXT( I ) )
         CALL MSG_OUT( ' ', '^TEXT', STATUS )
 1    CONTINUE

      END
```

To invoke this as a service routine, it is passed as an argument to NDF_HOUT, as follows:

```
      EXTERNAL NDF_HECHO

      ...

      CALL NDF_HOUT( INDF, IREC, NDF_HECHO, STATUS )
```

Note that it must be declared in a Fortran EXTERNAL statement.

NDF_HECHO provides a template if you wish to write your own service routine. By doing so, it is possible to perform other types of processing on *history* text. For instance, the following skeleton routine sets a logical value indicating whether the text of a *history* record contains a given sub-string (stored in the variable KEY), thus allowing a simple keyword search through *history* records to be performed:

```
      SUBROUTINE SEARCH( NLINES, TEXT, STATUS )
      INCLUDE 'SAE_PAR'

      CHARACTER KEY * ( 30 ), TEXT( NLINES ) * ( * )
      INTEGER I, NLINES, STATUS
      LOGICAL FOUND

      COMMON /KEYBLK/ KEY
      COMMON /FNDBLK/ FOUND

*  Initialise and check status.
      FOUND = .FALSE.
      IF ( STATUS .NE. SAI__OK ) RETURN

*  Search each text line in the record for the keyword.
      DO 1 I = 1, NLINES
        IF ( INDEX( TEXT( I ), KEY ) .NE. 0 ) THEN

*  Note when found.
          FOUND = .TRUE.
          GO TO 2
        END IF
 1    CONTINUE
 2    CONTINUE

      END
```

Note that additional data must be passed to and from such a service routine through global variables held in common blocks.

22.16 Modifying Past History

The NDF_ library does not allow you to re-write history. However, it does allow past *history* records to be deleted so that irrelevant information can be removed, usually to save space. The routine NDF_HPURG will perform this, as follows:

```
      INTEGER IREC1, IREC2
```

```
      ...

      CALL NDF_HPURG( INDF, IREC1, IREC2, STATUS )
```

Here, the *history* record numbers IREC1 and IREC2 specify a range of existing records to be deleted. Once these have been removed, any remaining records are re-numbered starting from 1.

It is recommended that users of NDF applications should normally invoke this routine via a utility application which records its own history information in the NDF. Thus, while the processing history may subsequently be incomplete, a record showing that deletion has occurred would still be present.

22.17 Naming an Application

Whenever a new *history* record is written by the NDF_ library, the name of the currently executing application is stored with it. The intention is that this should subsequently allow the software which was used to be identified as fully as possible. In practice, there are several possible sources from which the name of an application may be obtained, so a system of defaulting is used which operates as follows.

The first place in which the NDF_ library looks for an application name is the APPN argument of the NDF_HPUT routine. If this is not blank (and a new *history* record is being created by this routine), then the value supplied is used directly. This means that the writer of an application which explicitly writes *history* text via NDF_HPUT can also explicitly supply the name of the application as it is to appear in any new *history* record.

Whether it is a good idea to supply a name in this manner depends on the circumstances. The advantage is that the name given can be very specific (for example it might contain a version number for the application) and can easily be updated if the application is changed. Conversely, it makes it more difficult to re-use that software in another application and users may be confused if they invoke the application via a command whose name differs from that which is "hard-wired" into the software.

If the APPN argument to NDF_HPUT is blank, or if a new *history* record is being written by some other means (by the *default history recording* mechanism, for instance), the NDF_ library next looks to see if a "current" application name has previously been defined and, if so, it uses it. A current name may be declared at any time by calling the routine NDF_HAPPN, for example:

```
      CALL NDF_HAPPN( 'ADDSPEC Version 6.1-5', STATUS )
```

and may be revoked by calling the same routine with a blank first argument.

This method is best used in situations where several applications may be invoked by a single executable program. For instance, some software packages have a structure similar to the following, where a command is repeatedly obtained and the appropriate routine (application) is then invoked:

```
*  Loop to obtain commands.
 1    CONTINUE
      CALL GETCMD( CMD, STATUS )
```

```
*  Define the current application name.
      CALL NDF_HAPPN( CMD // ' (Mypack V3.4-2)', STATUS )

*  Invoke the appropriate routine.
      IF ( CMD .EQ. 'ADD' ) THEN
         CALL ADD( ... )
      ELSE IF ( CMD .EQ. 'SUB' ) THEN
         CALL SUB( ... )
      ELSE IF ( CMD .EQ. 'MUL' ) THEN
         CALL MUL( ... )
      ELSE IF ( CMD .EQ. 'DIV' ) THEN
         CALL DIV( ... )

      ...

      END IF

      ...

*  Return for the next command.
      GO TO 1
```

In such a situation, the NDF_ library has no way of telling when one application has finished and a new one is beginning, so it cannot generate separate names for them. This problem is solved by including a call to NDF_HAPPN before invoking each application, as above.

The final method of obtaining an application name is used only if both of the above methods have failed to produce a (non-blank) name. In this case, the NDF_ library will use a default method, whose precise details may depend on the software environment or operating system in use. For instance, the name of the currently executing file might be used.

The main disadvantage of allowing the NDF_ library to generate the application name itself is that it will not contain precise details (such as a software version number) and may not always be able to distinguish between separate applications invoked from within a single executable program, as in the earlier example. Nevertheless, it is recommended that this method should be adopted first, and either of the above methods substituted if it proves inadequate.

22.18 Ending an Application

One final consideration may apply if multiple applications are to be invoked from a single program (as in the example in §22.17) but some "global" NDF data structures are required to remain in use between applications. This is not a common requirement, but since *default history recording* is normally initiated only by the action of releasing an NDF, it will not occur in such a situation, so the global datasets will not automatically receive history information. To overcome this problem, a call to NDF_HEND may be used at the end of each application, as follows:

```
      CALL NDF_HEND( STATUS )
```

This notifies the NDF_ system that the current application has finished. It causes default history information to be written, if required, to all NDFs currently in use and ensures that new *history*

records will be created to contain any history information written subsequently (*i.e.* by the next application). NDF_HEND then revokes any "current application" name established via the NDF_HAPPN routine (see §22.17).

If its STATUS argument indicates an error condition, NDF_HEND also records the STATUS value and pending error message information in any suitable *history* components belonging to NDFs currently in use, as described in §22.12.

23 MISCELLANEOUS FACILITIES

23.1 Restricting Access via NDF Identifiers

Access restrictions may be imposed on any NDF identifier in order to constrain the operations which can be performed by NDF_ routines via that identifier. The act of disabling a particular type of access is performed by the routine NDF_NOACC. For instance:

```
      CALL NDF_NOACC( 'DELETE', INDF, STATUS )
```

would disable *delete* access for the identifier INDF. As a result, any subsequent attempt to delete the NDF via that identifier would fail, and the resulting error message would indicate that access had been disabled. Access restrictions imposed on NDF identifiers are propagated to all new identifiers derived from them (*e.g.* by cloning or creation of an NDF section). Once imposed, they cannot be revoked.

The following types of access may be disabled, either singly or in combination (by means of repeated calls to NDF_NOACC), in order to impose the corresponding restriction on NDF access:

'BOUNDS' – Disabling *bounds* access prevents the pixel-index bounds of a base NDF from being altered (*e.g.* with the routine NDF_SBND – §21.1). The pixel-index bounds of an NDF *section* can always be altered regardless of this access restriction, but the restriction will be propagated to any new identifier obtained from an NDF section via the routine NDF_BASE.

'DELETE' – Disabling *delete* access prevents the NDF from being deleted (*e.g.* with the routine NDF_DELET – §20.11).

'SHIFT' – Disabling *shift* access prevents pixel-index shifts from being applied to a base NDF (*e.g.* with the routine NDF_SHIFT – §21.2). Pixel-index shifts may always be applied to an NDF section regardless of this access restriction, but the restriction will be propagated to any new identifier obtained from an NDF section via the routine NDF_BASE.

'TYPE' – Disabling *type* access prevents the type of any NDF component from being altered (*e.g.* with the routine NDF_STYPE – §7.4).

'WRITE' – Disabling *write* access prevents new values from being written to any of the NDF's components. It also prevents the state of any of its components from being reset (*e.g.* with the routine NDF_RESET – §5.3).

In addition, specifying an access type of 'MODIFY' in a call to NDF_NOACC will disable all the forms of access described above.

You can enquire whether a specified type of access is available via any identifier by using the routine NDF_ISACC. For instance:

```
CALL NDF_ISACC( INDF, 'WRITE', ISACC, STATUS )
```

will return a logical value ISACC indicating whether *write* access is available for the identifier INDF.

23.2 Message System Routines

Several of the NDF_ routines are dedicated to generating components of messages, either for information or as part of an error report. These routines use the error and message reporting system (and the ERR_ and MSG_ routines) described in SUN/104.

The routine NDF_MSG may be used to assign the name of an NDF to a message token, so that references to NDFs can form part of a message, as follows:

```
      CALL NDF_MSG( 'NAME', INDF )
      CALL MSG_OUT( 'MESS_DEMO',
     :              'This NDF structure is called ^NAME', STATUS )
```

Here, 'NAME' is the message token name.

The routine NDF_CMSG may be used in a similar way to assign the value of an NDF character component to a message token, while NDF_ACMSG may be used to assign the value of an *axis* character component (such as an *axis label*) to a message token. Their use is illustrated in §6.1 and §19.6 respectively.

23.3 Tuning the NDF_ System

The routine NDF_TUNE is provided to allow various features of the NDF_ system to be configured for individual needs if the default behaviour is not appropriate. This process is referred to as *tuning* the system, and is performed as follows:

```
CALL NDF_TUNE( 0, 'WARN', STATUS )
```

Here, a new value of zero is specified for the *tuning parameter* called 'WARN'. The current setting of a tuning parameter may be determined using the related routine NDF_GTUNE which returns the parameter's value via its VALUE argument, as follows:

```
INTEGER VALUE

...

CALL NDF_GTUNE( 'WARN', VALUE, STATUS )
```

By using these two routines in pairs it is possible to determine the original setting of a tuning parameter, modify it locally, and then return it to its original value if necessary.

Each tuning parameter controls one aspect of the NDF_ system's behaviour according to the value which has been set for it. The tuning parameters currently available are as follows:

AUTO_HISTORY:
Controls whether to include an empty History component in NDFs created using NDF_NEW or NDF_CREAT. If the tuning parameter is zet to zero (the default), no History component will be included in the new NDFs. If the tuning parameter is zet non-zero, a History component will be added automatically to the new NDFs.

DOCVT:
Controls whether to convert foreign format data files to and from native NDF format for access (using the facilities described in SSN/20). If DOCVT is set to 1 (the default), and the other necessary steps described in SSN/20 have been taken, then such conversions will be performed whenever they are necessary to gain access to data stored in a foreign format. If DOCVT is set to 0, no such conversions will be attempted and all data will be accessed in native NDF format only.

The value of DOCVT may be changed at any time. It is the value current when a dataset is acquired by the NDF_ library (or a placeholder for a new dataset is created) which is significant.

KEEP:
Controls whether to retain a native format NDF copy of any foreign format data files which are accessed by the NDF_ library (and automatically converted using the facilities described in SSN/20). If KEEP is set to 0 (the default), then the results of converting foreign format data files will be stored in scratch filespace and deleted when no longer required. If KEEP is set to 1, the results of the conversion will instead be stored in permanent NDF data files in the default directory (such files will have the same name as the foreign file from which they are derived and a file type of '.sdf'). Setting KEEP to 1 may be useful if the same datasets are to be re-used, as it avoids having to convert them on each occasion.

The value of KEEP may be changed at any time. It is the value current when a foreign format file is first accessed by the NDF_ library which is significant.

ROUND:
Specifies the way in which floating-point values should be converted to integer during automatic type conversion. If ROUND is set to 1, then floating-point values are rounded to the nearest integer value. If ROUND is set to 0 (the default), floating-point values are truncated towards zero.

SECMAX:
Specifies the maximum number of pixels allowed in an NDF section, in units of mega-pixels. An error will be reported if an attempt is made to create a section containing more than this number of pixels. The primary purpose of this tuning parameter is to guard against accidental use of incorrect WCS units within a user-supplied section specified, that may result in huge sections being requested. The default value is 2147 mega-pixels, which is the largest value that can be held by a four byte integer.

SHCVT:

Controls whether diagnostic information is displayed to show the actions being taken to convert to and from foreign data formats (using the facilities described in SSN/20). If SHCVT is set to 1, then this information is displayed to assist in debugging external format conversion software whenever a foreign format file is accessed. If SHCVT is set to 0 (the default), this information does not appear and format conversion proceeds silently unless an error occurs.

TRACE:

Controls the reporting of additional error messages which may occasionally be useful for diagnosing internal problems within the NDF_ library. If TRACE is set to 1, then any error occurring within the NDF_ system will be accompanied by error messages indicating which internal routines have exited prematurely as a result. If TRACE is set to 0 (the default), this internal diagnostic information will not appear and only standard error messages will be produced.

WARN:

Controls the issuing of warning messages when certain non-fatal errors in the structure of NDF data objects are detected. If WARN is set to 1 (the default), then a warning message is issued. If WARN is set to 0, then no message is issued. In both cases normal execution continues and no STATUS value is set.

The value of tuning parameters may also be set by the user of NDF_ applications independently of the application itself. This is accomplished by defining an environment variable whose name is constructed by prefixing 'NDF_' to the name of the tuning parameter to be set. Thus, the shell command:

```
% setenv NDF_TRACE 1
```

would set the 'TRACE' tuning parameter to 1 by default for all subsequent NDF_ applications (*i.e.* over-riding the default described above). The application may, however, still modify the value itself.

Note that all tuning parameters set in this way must be given valid integer values. If the associated environment variable does not translate to a valid value, the default will remain unchanged.

24 COMPILING AND LINKING

In what follows, please note that the standalone and ADAM versions of the NDF_ library differ, in that those routines which use parameters (*i.e.* those listed in §C.14) are not available in the standalone version.

24.1 Standalone Applications

Standalone applications which use NDF_ routines may be linked by including execution of the command "ndf_link" on the compiler command line. Thus, to compile and link a Fortran application called "prog", the following might be used:

```
% f77 -I$STARLINK_DIR/include prog.f -L$STARLINK_DIR/lib `ndf_link` -o prog
```

while for an application written in C, you might use:

```
% cc prog.c -I$STARLINK_DIR/include -L$STARLINK_DIR/lib `ndf_link` -o prog
```

Note the use of backward quote characters, which cause the "ndf_link" command to be executed and its result substituted into the compiler command.

24.2 ADAM Applications

Users of the ADAM programming environment (SG/4) should use the "alink" command (SUN/144) to compile and link applications, and can access the NDF_ library by including execution of the command "ndf_link_adam" on the command line, as follows:

```
% alink adamprog.f `ndf_link_adam`
```

Again, note the use of backward quote characters. To build a program written in C (instead of Fortran), simply name the source file "prog.c", instead of "prog.f".

A EXAMPLE APPLICATIONS

This section contains a few simple example applications which demonstrate the use of NDF_ routines in real-life situations. These bring together many of the facilities which are described in relative isolation in other parts of the document.

All the following applications are written as ADAM A-tasks (see §24 for details of how to compile and link an A-task which calls NDF_ routines) and each may be extracted and used directly from the source of this document if required. On Starlink systems, this can be found in the file /star/docs/sun33.tex.

Readers who require a tutorial introduction to ADAM should consult SUN/101.

A.1 SHOW — Display the size of an NDF

This first example is trivial and simply serves to show the overall structure of an ADAM application which calls the NDF_ library. It outputs a message showing how many pixels there are in an NDF.

```
      SUBROUTINE SHOW( STATUS )
*+
*  Name:
*     SHOW

*  Purpose:
*     Show the size of an NDF.

*  Description:
*     This routine outputs a message showing how many pixels there are
*     in an NDF.

*  ADAM Parameters:
*     NDF = NDF (Read)
*        The NDF whose size is to be displayed.

*-

*  Type Definitions:
      IMPLICIT NONE              ! No implicit typing

*  Global Constants:
      INCLUDE 'SAE_PAR'          ! Standard SAE constants

*  Status:
      INTEGER STATUS             ! Global status

*  Local Variables:
      INTEGER INDF               ! NDF identifier
      INTEGER NPIX               ! Number of NDF pixels

*.
```

```
*  Check inherited global status.
      IF ( STATUS .NE. SAI__OK ) RETURN

*  Obtain the input NDF and enquire its size.
      CALL NDF_ASSOC( 'NDF', 'READ', INDF, STATUS )
      CALL NDF_SIZE( INDF, NPIX, STATUS )

*  Display the size.
      CALL MSG_SETI( 'NPIX', NPIX )
      CALL MSG_OUT( 'SHOW_SIZE', 'This NDF has ^NPIX pixels.', STATUS )

*  Annul the NDF identifier.
      CALL NDF_ANNUL( INDF, STATUS )

      END
```

The following is an example ADAM interface file (show.ifl) for the application above.

```
interface SHOW

   parameter NDF                 # NDF to be inspected
      position 1
      prompt   'NDF data structure'
   endparameter

endinterface
```

A.2 SETTITLE — Assign a New NDF Title

The following example is a simple application which sets a new *title* for an existing NDF. Note the use of 'UPDATE' access, since an existing NDF is being modified rather than creating a new one.

```
      SUBROUTINE SETTITLE( STATUS )
*+
*  Name:
*     SETTITLE

*  Purpose:
*     Set a new title for an NDF data structure.

*  Description:
*     This routine sets a new value for the title component of an
*     existing NDF data structure. The NDF is accessed in update mode
*     and any pre-existing title is over-written with a new value.
*     Alternatively, if a "null" value (!) is given for the TITLE
*     parameter, then the NDF's title will be erased.

*  ADAM Parameters:
*     NDF = NDF (Read and Write)
*        The NDF data structure whose title is to be modified.
*     TITLE = LITERAL (Read)
*        The value to be assigned to the NDF's title component. [!]

*-

*  Type Definitions:
      IMPLICIT NONE              ! No implicit typing

*  Global Constants:
      INCLUDE 'SAE_PAR'          ! Standard SAE constants

*  Status:
      INTEGER STATUS             ! Global status

*  Local Variables:
      INTEGER NDF                ! NDF identifier

*.

*  Check inherited global status.
      IF ( STATUS .NE. SAI__OK ) RETURN

*  Obtain an identifier for the NDF to be modified.
      CALL NDF_ASSOC( 'NDF', 'UPDATE', NDF, STATUS )

*  Reset any existing title.
      CALL NDF_RESET( NDF, 'Title', STATUS )

*  Obtain a new title.
      CALL NDF_CINP( 'TITLE', NDF, 'Title', STATUS )
```

```
*  Annul the NDF identifier.
      CALL NDF_ANNUL( NDF, STATUS )

      END
```

The following is an example ADAM interface file (settitle.ifl) for the application above.

```
interface SETTITLE

   parameter NDF                 # NDF to be modified
      position 1
      prompt   'Data structure'
   endparameter

   parameter TITLE               # New title value
      position 2
      type     'LITERAL'
      prompt   'New NDF title'
   endparameter

endinterface
```

A.3 GETMAX — Obtain the Maximum Pixel Value

The following application calculates and displays the maximum pixel value in an NDF's *data* array. It is typical of a class of applications which read a single NDF as input, but do not produce any output data structure.

In this example, the choice has been made to handle all values using single-precision (_REAL) arithmetic and not to handle *bad* pixel values at all. Strictly speaking, the call to NDF_MBAD to check for the presence of *bad* pixels is not essential, but it does help by producing an error message if someone should inadvertently use this program on data which does contain *bad* pixels.

```
      SUBROUTINE GETMAX( STATUS )
*+
*  Name:
*     GETMAX

*  Purpose:
*     Obtain the maximum pixel value.

*  Description:
*     This routine finds the maximum pixel value in the data array of
*     an NDF and displays the result.

*  ADAM Parameters:
*     NDF = NDF (Read)
*        The NDF data structure whose data array is to be examined.

*  Implementation Status:
*     This routine deliberately does not handle NDFs whose data arrays
*     contain bad pixels. Real arithmetic is used to compute the
*     maximum.

*-

*  Type Definitions:
      IMPLICIT NONE              ! No implicit typing

*  Global Constants:
      INCLUDE 'SAE_PAR'          ! Standard SAE constants

*  Status:
      INTEGER STATUS             ! Global status

*  Local Variables:
      INTEGER EL                 ! Number of mapped pixels
      INTEGER INDF               ! NDF identifier
      INTEGER PNTR( 1 )          ! Pointer to mapped values
      LOGICAL BAD                ! Bad pixels present? (junk variable)
      REAL HIGH                  ! Maximum pixel value

*.

*  Check inherited global status.
```

```
      IF ( STATUS .NE. SAI__OK ) RETURN

*  Obtain the input NDF and map its data array as _REAL values for
*  reading.
      CALL NDF_ASSOC( 'NDF', 'READ', INDF, STATUS )
      CALL NDF_MAP( INDF, 'Data', '_REAL', 'READ', PNTR, EL, STATUS )

*  Check that there are no bad pixels present.
      CALL NDF_MBAD( .FALSE., INDF, INDF, 'Data', .TRUE., BAD, STATUS )

*  Find the maximum pixel value and display the result.
      CALL MAXIT( EL, %VAL( PNTR( 1 ) ), HIGH, STATUS )
      CALL MSG_SETR( 'HIGH', HIGH )
      CALL MSG_OUT( 'GETMAX_HIGH', '   Maximum value is ^HIGH', STATUS )

*  Annul the NDF identifier.
      CALL NDF_ANNUL( INDF, STATUS )

      END

      SUBROUTINE MAXIT( EL, ARRAY, HIGH, STATUS )
*+
*  Name:
*     MAXIT

*  Purpose:
*     Find the maximum value in a real array.

*  Invocation:
*     CALL MAXIT( EL, ARRAY, HIGH, STATUS )

*  Description:
*     The routine returns the maximum element value in a real array.

*  Arguments:
*     EL = INTEGER (Given)
*        Number of array elements.
*     ARRAY( EL ) = REAL (Given)
*        The real array.
*     HIGH = REAL (Returned)
*        Maximum element value.
*     STATUS = INTEGER (Given and Returned)
*        The global status.

*-

*  Type Definitions:
      IMPLICIT NONE              ! No implicit typing

*  Global Constants:
      INCLUDE 'SAE_PAR'          ! Standard SAE constants

*  Arguments Given:
      INTEGER EL
```

```
      REAL ARRAY( * )

*  Arguments Returned:
      REAL HIGH

*  Status:
      INTEGER STATUS             ! Global status

*  Local Variables:
      INTEGER I                  ! Loop counter

*.

*  Check inherited global status.
      IF ( STATUS .NE. SAI__OK ) RETURN

*  Find the maximum array value.
      HIGH = ARRAY( 1 )
      DO 1 I = 2, EL
         IF ( ARRAY( I ) .GT. HIGH ) HIGH = ARRAY( I )
 1    CONTINUE

      END
```

The following is an example ADAM interface file (getmax.ifl) for the application above.

```
interface GETMAX

   parameter NDF                 # NDF to be examined
      position 1
      prompt   'Data structure'
   endparameter

endinterface
```

A.4 GETSUM — Sum the Pixel Values

This application is a logical extension of the previous one, except that it sums the pixel values in an NDF's data array, rather than finding the maximum pixel value. In this example, however, we first check to determine whether or not there may be *bad* pixel values in the input NDF, and then adapt the algorithm to accommodate either case. Any *bad* pixels are excluded from the result.

Simple error reporting is also introduced in this example; an error report is generated if the input data array does not contain any good (*i.e.* non-*bad*) pixels.

```
      SUBROUTINE GETSUM( STATUS )
*+
*  Name:
*     GETSUM

*  Purpose:
*     Sum the pixels in an NDF's data array.

*  Description:
*     This routine sums the values of the pixels in an NDF's data array
*     and displays the result. Any bad pixels which may be present are
*     excluded from the sum.

*  ADAM Parameters:
*     NDF = NDF (Read)
*        The NDF data structure whose data array is to be examined.

*  Implementation Status:
*     This routine can handle data with or without bad pixels (and
*     hence can also handle a quality array if present). Real
*     arithmetic is used for forming the pixel sum.

*-

*  Type Definitions:
      IMPLICIT NONE              ! No implicit typing

*  Global Constants:
      INCLUDE 'SAE_PAR'          ! Standard SAE constants

*  Status:
      INTEGER STATUS             ! Global status

*  Local Variables:
      INTEGER EL                 ! Number of mapped pixels
      INTEGER INDF               ! NDF identifier
      INTEGER NGOOD              ! Number of good pixels
      INTEGER PNTR( 1 )          ! Pointer to mapped values
      LOGICAL BAD                ! Bad pixel present?
      REAL SUM                   ! Pixel sum

*.
```

```
*  Check inherited global status.
      IF ( STATUS .NE. SAI__OK ) RETURN

*  Obtain the input NDF and map its data array as _REAL values for
*  reading.
      CALL NDF_ASSOC( 'NDF', 'READ', INDF, STATUS )
      CALL NDF_MAP( INDF, 'Data', '_REAL', 'READ', PNTR, EL, STATUS )

*  See if bad pixel values are present.
      CALL NDF_BAD( INDF, 'Data', .FALSE., BAD, STATUS )

*  Sum the pixel values and display the result.
      CALL SUMIT( BAD, EL, %VAL( PNTR( 1 ) ), SUM, NGOOD, STATUS )
      IF ( NGOOD .GT. 0 ) THEN
         CALL MSG_SETR( 'SUM', SUM )
         CALL MSG_OUT( 'GETSUM_SUM',
     :                 '   Sum of pixels is ^SUM', STATUS )

*  Report an error if there are no good pixels present.
      ELSE
         STATUS = SAI__ERROR
         CALL NDF_MSG( 'NDF', INDF )
         CALL ERR_REP( 'GETSUM_ALLBAD',
     :   'GETSUM: All the data pixels in the NDF ^NDF are bad.',
     :   STATUS )
      END IF

*  Annul the NDF identifier.
      CALL NDF_ANNUL( INDF, STATUS )

      END

      SUBROUTINE SUMIT( BAD, EL, ARRAY, SUM, NGOOD, STATUS )
*+
*  Name:
*     SUMIT

*  Purpose:
*     Sum the elements of a real array, allowing for bad values.

*  Invocation:
*     CALL SUMIT( BAD, EL, ARRAY, SUM, NGOOD, STATUS )

*  Description:
*     The routine returns the sum of the elements of a real array,
*     excluding any which have the bad value.

*  Arguments:
*     BAD = LOGICAL (Given)
*        Whether bad pixel values may be present.
*     EL = INTEGER (Given)
*        Number of array elements.
*     ARRAY( EL ) = REAL (Given)
*        The real array.
```

```
*     SUM = REAL (Returned)
*        Sum of the elements.
*     NGOOD = INTEGER (Returned)
*        Number of good (non-bad) elements.
*     STATUS = INTEGER (Given and Returned)
*        The global status.

*-

*  Type Definitions:
      IMPLICIT NONE              ! No implicit typing

*  Global Constants:
      INCLUDE 'SAE_PAR'          ! Standard SAE constants
      INCLUDE 'PRM_PAR'          ! Define the VAL__BADR constant

*  Arguments Given:
      LOGICAL BAD
      INTEGER EL
      REAL ARRAY( * )

*  Arguments Returned:
      REAL SUM
      INTEGER NGOOD

*  Status:
      INTEGER STATUS             ! Global status

*  Local Variables:
      INTEGER I                  ! Loop counter

*.

*  Check inherited global status.
      IF ( STATUS .NE. SAI__OK ) RETURN

*  If there are no bad values, simply sum the array elements.
      IF ( .NOT. BAD ) THEN
         SUM = 0.0
         NGOOD = EL
         DO 1 I = 1, EL
            SUM = SUM + ARRAY( I )
 1       CONTINUE

*  Otherwise, test each element before using it.
      ELSE
         SUM = 0.0
         NGOOD = 0
         DO 2 I = 1, EL
            IF ( ARRAY( I ) .NE. VAL__BADR ) THEN
               SUM = SUM + ARRAY( I )
               NGOOD = NGOOD + 1
            END IF
 2       CONTINUE
```

```
      END IF

      END
```

The following is an example ADAM interface file (getsum.ifl) for the application above.

```
interface GETSUM

   parameter NDF                 # NDF to be examined
      position 1
      prompt   'Data structure'
   endparameter

endinterface
```

A.5 READIMG — Read an image into an NDF

The following is a simple example of how to create a new NDF data structure from scratch, in this case starting with image data read from an unformatted sequential Fortran file. This is typical of how the NDF-based half of a format conversion application (designed to read data from another format into an NDF) might look. This example could be modified to read other formats by appropriately replacing the routine which reads the data from the file.

In this example, use has been made of the FIO_ package (SUN/143) to allocate a Fortran I/O unit. Some moderately elaborate error reporting is also illustrated; this gives helpful error messages in response to I/O errors and a final contextual report at the end of the application.

```
      SUBROUTINE READIMG( STATUS )
*+
*  Name:
*     READIMG

*  Purpose:
*     Read an image into an NDF.

*  Description:
*     This routine reads a 2-dimensional real image into an NDF data
*     structure from an unformatted sequential Fortran file. The image
*     data are assumed to be stored one line per record in the file.

*  ADAM Parameters:
*     FILE = LITERAL (Read)
*        Name of the input file.
*     NDF = NDF (Write)
*        The output NDF data structure.
*     NX = INTEGER (Read)
*        Number of pixels per image line.
*     NY = INTEGER (Read)
*        Number of lines in the image.

*-

*  Type Definitions:
      IMPLICIT NONE              ! No implicit typing

*  Global Constants:
      INCLUDE 'SAE_PAR'          ! Standard SAE constants
      INCLUDE 'FIO_PAR'          ! Define FIO__SZFNM constant

*  Status:
      INTEGER STATUS             ! Global status

*  Local Variables:
      CHARACTER * ( FIO__SZFNM ) FILE ! Input file name
      INTEGER DIM( 2 )           ! Image dimension sizes
      INTEGER EL                 ! Number of mapped values
      INTEGER INDF               ! NDF identifier
      INTEGER IOERR              ! I/O error status
      INTEGER IOUNIT             ! Fortran I/O unit number
```

```
      INTEGER PNTR( 1 )          ! Pointer to mapped values

*.

*  Check inherited global status.
      IF ( STATUS .NE. SAI__OK ) RETURN

*  Obtain the name of the input file and allocate an I/O unit on which
*  to open it.
      CALL PAR_GETOC( 'FILE', FILE, STATUS )
      CALL FIO_GUNIT( IOUNIT, STATUS )
      IF ( STATUS .NE. SAI__OK ) GO TO 99

*  Open the file, trapping and reporting any errors.
      OPEN( FILE = FILE, UNIT = IOUNIT, STATUS = 'OLD',
     :      FORM = 'UNFORMATTED', IOSTAT = IOERR )
      IF ( IOERR .NE. 0 ) THEN
         STATUS = SAI__ERROR
         CALL ERR_FIOER( 'MESSAGE', IOERR )
         CALL FIO_REP( IOUNIT, FILE, IOERR,
     :   'Unable to open file ^FNAME on Fortran unit ^UNIT - ' //
     :   '^MESSAGE', STATUS )
         GO TO 99
      END IF

*  Obtain the dimension sizes of the image to be read and check the
*  values obtained for validity.
      CALL PAR_GET0I( 'NX', DIM( 1 ), STATUS )
      CALL PAR_GET0I( 'NY', DIM( 2 ), STATUS )
      IF ( STATUS .NE. SAI__OK ) GO TO 99
      IF ( ( DIM( 1 ) .LT. 1 ) .OR. ( DIM( 2 ) .LT. 1 ) ) THEN
         STATUS = SAI__ERROR
         CALL ERR_REP( 'READIMG_BADDIM',
     :   'Image dimensions must be positive.', STATUS )
         GO TO 99
      END IF

*  Create an output NDF of the correct size and map its data array as
*  _REAL values for writing.
      CALL NDF_CREP( 'NDF', '_REAL', 2, DIM, INDF, STATUS )
      CALL NDF_MAP( INDF, 'Data', '_REAL', 'WRITE', PNTR, EL, STATUS )

*  Read the image values from the input file into the mapped array.
      CALL READIT( IOUNIT, DIM( 1 ), DIM( 2 ), %VAL( PNTR( 1 ) ),
     :             STATUS )

*  Annul the NDF identifier, close the input file and deallocate the I/O
*  unit.
 99   CONTINUE
      CALL NDF_ANNUL( INDF, STATUS )
      CLOSE( UNIT = IOUNIT )
      CALL FIO_PUNIT( IOUNIT, STATUS )

*  If an error occurred, then report contextual information.
```

```
      IF ( STATUS .NE. SAI__OK ) THEN
         CALL ERR_REP( 'READIMG_ERR',
     :   'READIMG: Error reading an image into an NDF from a ' //
     :   'Fortran file.', STATUS )
      END IF

      END

      SUBROUTINE READIT( IOUNIT, NX, NY, ARRAY, STATUS )
*+
*  Name:
*     READIT

*  Purpose:
*     Read an image from a file.

*  Invocation:
*     CALL READIT( IOUNIT, NX, NY, ARRAY, STATUS )

*  Description:
*     The routine reads a real image from an unformatted sequential
*     Fortran file, one image line per record.

*  Arguments:
*     IOUNIT = INTEGER (Given)
*        The Fortran I/O unit on which to read the (previously opened)
*        file.
*     NX = INTEGER (Given)
*        Number of pixels per image line.
*     NY = INTEGER (Given)
*        Number of lines in the image.
*     ARRAY( NX, NY ) = REAL (Returned)
*        The image array to be read.
*     STATUS = INTEGER (Given and Returned)
*        The global status.

*-

*  Type Definitions:
      IMPLICIT NONE             ! No implicit typing

*  Global Constants:
      INCLUDE 'SAE_PAR'         ! Standard SAE constants
      INCLUDE 'FIO_PAR'         ! Define FIO__SZFNM constant

*  Arguments Given:
      INTEGER IOUNIT
      INTEGER NX
      INTEGER NY

*  Arguments Returned:
      REAL ARRAY( NX, NY )

*  Status:
```

```
      INTEGER STATUS             ! Global status

*  Local Variables:
      CHARACTER * ( FIO__SZFNM ) FILE ! File name
      INTEGER IGNORE             ! Enquire status (ignored)
      INTEGER IOERR              ! I/O error status
      INTEGER IX                 ! Loop counter for image pixels
      INTEGER IY                 ! Loop counter for image lines

*.

*  Check inherited global status.
      IF ( STATUS .NE. SAI__OK ) RETURN

*  Read each line of the image from the file, trapping any errors.
      DO 1 IY = 1, NY
         READ( IOUNIT, IOSTAT = IOERR ) ( ARRAY( IX, IY ), IX= 1, NX )

*  If an error occurred, then make a helpful error report and abort.
         IF ( IOERR .NE. 0 ) THEN
            STATUS = SAI__ERROR
            CALL ERR_FIOER( 'MESSAGE', IOERR )
            CALL FIO_REP( IOUNIT, ' ', IOERR,
     :      'Error reading file ^FNAME on Fortran unit '^UNIT - ' //
     :      '^MESSAGE', STATUS )
            GO TO 99
         END IF
 1    CONTINUE

*  Jump to here if an error occurs.
 99   CONTINUE
      END
```

The following is an example ADAM interface file (readimg.ifl) for the application above.

```
interface READIMG

   parameter FILE              # Input file name
      position 1
      type     LITERAL
      prompt   'Input file'
   endparameter

   parameter NDF               # Output NDF
      position 4
      prompt   'Output NDF'
   endparameter

   parameter NX                # Number of pixels per line
      position 2
      type     _INTEGER
      prompt   'X dimension of image'
   endparameter
```

```
   parameter NY                  # Number of lines in image
      position 3
      type     _INTEGER
      prompt   'Y dimension of image'
   endparameter

endinterface
```

A.6 ZAPPIX — "Zap" Prominent Pixels in an Image

The following example is based around a simple algorithm which detects prominent pixels (*e.g.* data spikes) in a 2-dimensional image and replaces them with *bad* pixels. It is typical of applications which take a single NDF as input and produce a new NDF with the same size as output. It illustrates the use of *propagation* (NDF_PROP) in producing the new output NDF using the input as a template. Note that this application modifies the *data* array but does not handle the *variance* array, which will therefore become invalid and is not propagated.

This example also illustrates how *bad* pixels might be handled in a reasonably realistic image-processing algorithm. No attempt is made here to distinguish cases where *bad* pixels are present from those where they are not, and we do not really know afterwards if there are any *bad* pixels in the output image (although a check for this could easily be added). The output *bad*-pixel flag is therefore left with its default value of .TRUE..

```
      SUBROUTINE ZAPPIX( STATUS )
*+
*  Name:
*     ZAPPIX

*  Purpose:
*     Zap prominent pixels.

*  Description:
*     This routine "zaps" prominent pixels in a 2-dimensional image
*     (stored in the data array of an NDF). It searches for pixels
*     which deviate by more than a specified amount from the mean of
*     their nearest neighbours, and replaces them with bad pixels.

*  ADAM Parameters:
*     IN = NDF (Read)
*        Input NDF data structure.
*     OUT = NDF (Write)
*        The output NDF data structure.
*     THRESH = _REAL (Read)
*        Threshold for zapping pixels; pixels will be set bad if they
*        deviate by more than this amount from the mean of their
*        nearest neighbours (the absolute value of THRESH is used).

*  Implementation Status:
*     This routine correctly handles bad pixels but does not handle NDF
*     variance arrays. Real arithmetic is used.

*-

*  Type Definitions:
      IMPLICIT NONE              ! No implicit typing

*  Global Constants:
      INCLUDE 'SAE_PAR'          ! Standard SAE constants

*  Status:
      INTEGER STATUS             ! Global status
```

```
*  Local Variables:
      INTEGER DIM( 2 )            ! Image dimension sizes
      INTEGER EL                  ! Number of mapped values
      INTEGER INDF1               ! Input NDF identifier
      INTEGER INDF2               ! Output NDF identifier
      INTEGER NDIM                ! Number of NDF dimensions (junk)
      INTEGER PNTR1( 1 )          ! Pointer to mapped input values
      INTEGER PNTR2( 1 )          ! Pointer to mapped output values
      REAL THRESH                 ! Threshold for zapping pixels

*.

*  Check inherited global status.
      IF ( STATUS .NE. SAI__OK ) RETURN

*  Begin an NDF context.
      CALL NDF_BEGIN

*  Obtain the input NDF and obtain its first two dimension sizes.
      CALL NDF_ASSOC( 'IN', 'READ', INDF1, STATUS )
      CALL NDF_DIM( INDF1, 2, DIM, NDIM, STATUS )

*  Obtain a threshold value.
      CALL PAR_GETOR( 'THRESH', THRESH, STATUS )

*  Create an output NDF based on the input one. Propagate the AXIS,
*  QUALITY and UNITS components.
      CALL NDF_PROP( INDF1, 'Axis,Quality,Units', 'OUT', INDF2, STATUS )

*  Map the input and output data arrays for reading and writing
*  respectively.
      CALL NDF_MAP( INDF1, 'Data', '_REAL', 'READ', PNTR1, EL, STATUS )
      CALL NDF_MAP( INDF2, 'Data', '_REAL', 'WRITE', PNTR2, EL, STATUS )

*  Process the input array, writing the new values to the output array.
      CALL ZAPIT( ABS( THRESH ), DIM( 1 ), DIM( 2 ), %VAL( PNTR1( 1 ) ),
     :            %VAL( PNTR2( 1 ) ), STATUS )

*  End the NDF context (this cleans everything up).
      CALL NDF_END( STATUS )

*  If an error occurred, then report a contextual message.
      IF ( STATUS .NE. SAI__OK ) THEN
         CALL ERR_REP( 'ZAPPIX_ERR',
     :   'ZAPPIX: Error zapping prominent pixels in an image.',
     :   STATUS )
      END IF

      END

      SUBROUTINE ZAPIT( THRESH, NX, NY, A, B, STATUS )
*+
*  Name:
```

```
*     ZAPIT

*  Purpose:
*     Zap prominent pixels in an image.

*  Invocation:
*     CALL ZAPIT( THRESH, NX, NY, A, B, STATUS )

*  Description:
*     The routine finds all pixels in a 2-dimensional image which
*     deviate by more than a specified amount from the mean of their
*     nearest neighbours and replaces them with the bad pixel value.
*     Bad pixels in the input image are correctly handled.

*  Arguments:
*     THRESH = REAL (Given)
*        The threshold for zapping pixels.
*     NX = INTEGER (Given)
*        X dimension of image.
*     NY = INTEGER (Given)
*        Y dimension of image.
*     A( NX, NY ) = REAL (Given)
*        The input image.
*     B( NX, NY ) = REAL (Returned)
*        The output image.
*     STATUS = INTEGER (Given and Returned)
*        The global status.

*-

*  Type Definitions:
      IMPLICIT NONE             ! No implicit typing

*  Global Constants:
      INCLUDE 'SAE_PAR'         ! Standard SAE constants
      INCLUDE 'PRM_PAR'         ! Define VAL__BADR constant

*  Arguments Given:
      REAL THRESH
      INTEGER NX
      INTEGER NY
      REAL A( NX, NY )

*  Arguments Returned:
      REAL B( NX, NY )

*  Status:
      INTEGER STATUS            ! Global status

*  Local Variables:
      INTEGER IIX               ! Loop counter for neighbours
      INTEGER IIY               ! Loop counter for neighbours
      INTEGER IX                ! Loop counter for image pixels
      INTEGER IY                ! Loop counter for image pixels
```

```
      INTEGER N                   ! Number of good neighbours
      REAL DIFF                   ! Deviation from mean of neighbours
      REAL S                      ! Sum of good neighbours

*.

*  Check inherited global status.
      IF ( STATUS .NE. SAI__OK ) RETURN

*  Loop through all the pixels in the image.
      DO 4 IY = 1, NY
         DO 3 IX = 1, NX

*  If the input pixel is bad, then so is the output pixel.
            IF ( A( IX, IY ) .EQ. VAL__BADR ) THEN
               B( IX, IY ) = VAL__BADR

*  Otherwise, loop to find the average of the nearest neighbours.
            ELSE
               S = 0.0
               N = 0
               DO 2 IIY = MAX( 1, IY - 1 ), MIN( NY, IY + 1 )
                  DO 1 IIX = MAX( 1, IX - 1 ), MIN( NX, IX + 1 )

*  Only count neighbours which are not bad themselves.
                     IF ( A( IIX, IIY ) .NE. VAL__BADR ) THEN
                        S = S + A( IIX, IIY )
                        N = N + 1
                     END IF
 1                CONTINUE
 2             CONTINUE

*  If all the neighbours were bad, then just copy the central pixel.
               IF ( N .EQ. 0 ) THEN
                  B( IX, IY ) = A( IX, IY )

*  Otherwise, see if the central pixel deviates by more than THRESH from
*  the average. If not, copy it. If so, set it bad.
               ELSE
                  DIFF = A( IX, IY ) - ( S / REAL( N ) )
                  IF ( ABS( DIFF ) .LE. THRESH ) THEN
                     B( IX, IY ) = A( IX, IY )
                  ELSE
                     B( IX, IY ) = VAL__BADR
                  END IF
               END IF
            END IF
 3       CONTINUE
 4    CONTINUE

      END
```

The following is an example ADAM interface file (zappix.ifl) for the application above.

```
interface ZAPPIX

   parameter IN                 # Input NDF
      position 1
      prompt   'Input NDF'
   endparameter

   parameter OUT                # Output NDF
      position 3
      prompt   'Output NDF'
   endparameter

   parameter THRESH             # Zapping threshold
      position 2
      type     _REAL
      prompt   'Threshold'
   endparameter

endinterface
```

A.7 ADD — Add Two NDF Data Structures

The following application adds two NDF data structures pixel-by-pixel. It is a fairly sophisticated "add" application, which will handle both the *data* and *variance* components, as well as coping with NDFs of any shape and data type. A much simpler example is given in §2.4.

```
      SUBROUTINE ADD( STATUS )
*+
*  Name:
*     ADD

*  Purpose:
*     Add two NDF data structures.

*  Description:
*     This routine adds two NDF data structures pixel-by-pixel to produce
*     a new NDF.

*  ADAM Parameters:
*     IN1 = NDF (Read)
*        First NDF to be added.
*     IN2 = NDF (Read)
*        Second NDF to be added.
*     OUT = NDF (Write)
*        Output NDF to contain the sum of the two input NDFs.
*     TITLE = LITERAL (Read)
*        Value for the title of the output NDF. A null value will cause
*        the title of the NDF supplied for parameter IN1 to be used
*        instead. [!]

*-

*  Type Definitions:
      IMPLICIT NONE              ! No implicit typing

*  Global Constants:
      INCLUDE 'SAE_PAR'          ! Standard SAE constants
      INCLUDE 'NDF_PAR'          ! NDF_ public constants

*  Status:
      INTEGER STATUS             ! Global status

*  Local Variables:
      CHARACTER * ( 13 ) COMP    ! NDF component list
      CHARACTER * ( NDF__SZFTP ) DTYPE ! Type for output components
      CHARACTER * ( NDF__SZTYP ) ITYPE ! Numeric type for processing
      INTEGER EL                 ! Number of mapped elements
      INTEGER IERR               ! Position of first error (dummy)
      INTEGER NDF1               ! Identifier for 1st NDF (input)
      INTEGER NDF2               ! Identifier for 2nd NDF (input)
      INTEGER NDF3               ! Identifier for 3rd NDF (output)
      INTEGER NERR               ! Number of errors
      INTEGER PNTR1( 2 )         ! Pointers to 1st NDF mapped arrays
      INTEGER PNTR2( 2 )         ! Pointers to 2nd NDF mapped arrays
```

```
      INTEGER PNTR3( 2 )         ! Pointers to 3rd NDF mapped arrays
      LOGICAL BAD                ! Need to check for bad pixels?
      LOGICAL VAR1               ! Variance component in 1st input NDF?
      LOGICAL VAR2               ! Variance component in 2nd input NDF?

*.

*  Check inherited global status.
      IF ( STATUS .NE. SAI__OK ) RETURN

*  Begin an NDF context.
      CALL NDF_BEGIN

*  Obtain identifiers for the two input NDFs.
      CALL NDF_ASSOC( 'IN1', 'READ', NDF1, STATUS )
      CALL NDF_ASSOC( 'IN2', 'READ', NDF2, STATUS )

*  Trim their pixel-index bounds to match.
      CALL NDF_MBND( 'TRIM', NDF1, NDF2, STATUS )

*  Create a new output NDF based on the first input NDF. Propagate the
*  axis and quality components, which are not changed. This program
*  does not support the units component.
      CALL NDF_PROP( NDF1, 'Axis,Quality', 'OUT', NDF3, STATUS )

*  See if a variance component is available in both input NDFs and
*  generate an appropriate list of input components to be processed.
      CALL NDF_STATE( NDF1, 'Variance', VAR1, STATUS )
      CALL NDF_STATE( NDF2, 'Variance', VAR2, STATUS )
      IF ( VAR1 .AND. VAR2 ) THEN
         COMP = 'Data,Variance'
      ELSE
         COMP = 'Data'
      END IF

*  Determine which numeric type to use to process the input arrays and
*  set an appropriate type for the corresponding output arrays. This
*  program supports integer, real and double-precision arithmetic.
      CALL NDF_MTYPE( '_INTEGER,_REAL,_DOUBLE',
     :                NDF1, NDF2, COMP, ITYPE, DTYPE, STATUS )
      CALL NDF_STYPE( DTYPE, NDF3, COMP, STATUS )

*  Map the input and output arrays.
      CALL NDF_MAP( NDF1, COMP, ITYPE, 'READ', PNTR1, EL, STATUS )
      CALL NDF_MAP( NDF2, COMP, ITYPE, 'READ', PNTR2, EL, STATUS )
      CALL NDF_MAP( NDF3, COMP, ITYPE, 'WRITE', PNTR3, EL, STATUS )

*  Merge the bad pixel flag values for the input data arrays to see if
*  checks for bad pixels are needed.
      CALL NDF_MBAD( .TRUE., NDF1, NDF2, 'Data', .FALSE., BAD, STATUS )

*  Select the appropriate routine for the data type being processed and
*  add the data arrays.
      IF ( STATUS .EQ. SAI__OK ) THEN
```

```
            IF ( ITYPE .EQ. '_INTEGER' ) THEN
               CALL VEC_ADDI( BAD, EL, %VAL( PNTR1( 1 ) ),
     :                        %VAL( PNTR2( 1 ) ), %VAL( PNTR3( 1 ) ),
     :                        IERR, NERR, STATUS )

            ELSE IF ( ITYPE .EQ. '_REAL' ) THEN
               CALL VEC_ADDR( BAD, EL, %VAL( PNTR1( 1 ) ),
     :                        %VAL( PNTR2( 1 ) ), %VAL( PNTR3( 1 ) ),
     :                        IERR, NERR, STATUS )

            ELSE IF ( ITYPE .EQ. '_DOUBLE' ) THEN
               CALL VEC_ADDD( BAD, EL, %VAL( PNTR1( 1 ) ),
     :                        %VAL( PNTR2( 1 ) ), %VAL( PNTR3( 1 ) ),
     :                        IERR, NERR, STATUS )
            END IF

*  Flush any messages resulting from numerical errors.
            IF ( STATUS .NE. SAI__OK ) CALL ERR_FLUSH( STATUS )
         END IF

*  See if there may be bad pixels in the output data array and set the
*  output bad pixel flag value accordingly.
         BAD = BAD .OR. ( NERR .NE. 0 )
         CALL NDF_SBAD( BAD, NDF3, 'Data', STATUS )

*  If variance arrays are also to be processed (i.e. added), then see
*  if bad pixels may be present.
         IF ( VAR1 .AND. VAR2 ) THEN
            CALL NDF_MBAD( .TRUE., NDF1, NDF2, 'Variance', .FALSE., BAD,
     :                     STATUS )

*  Select the appropriate routine to add the variance arrays.
            IF (STATUS .EQ. SAI__OK ) THEN
               IF ( ITYPE .EQ. '_INTEGER' ) THEN
                  CALL VEC_ADDI( BAD, EL, %VAL( PNTR1( 2 ) ),
     :                           %VAL( PNTR2( 2 ) ), %VAL( PNTR3( 2 ) ),
     :                           IERR, NERR, STATUS )

               ELSE IF ( ITYPE .EQ. '_REAL' ) THEN
                  CALL VEC_ADDR( BAD, EL, %VAL( PNTR1( 2 ) ),
     :                           %VAL( PNTR2( 2 ) ), %VAL( PNTR3( 2 ) ),
     :                           IERR, NERR, STATUS )

               ELSE IF ( ITYPE .EQ. '_DOUBLE' ) THEN
                  CALL VEC_ADDD( BAD, EL, %VAL( PNTR1( 2 ) ),
     :                           %VAL( PNTR2( 2 ) ), %VAL( PNTR3( 2 ) ),
     :                           IERR, NERR, STATUS )
               END IF

*  Flush any messages resulting from numerical errors.
               IF ( STATUS .NE. SAI__OK ) CALL ERR_FLUSH( STATUS )
            END IF

*  See if bad pixels may be present in the output variance array and
```

```
*  set the bad pixel flag accordingly.
         BAD = BAD .OR. ( NERR .NE. 0 )
         CALL NDF_SBAD( BAD, NDF3, 'Variance', STATUS )
      END IF

*  Obtain a new title for the output NDF.
      CALL NDF_CINP( 'TITLE', NDF3, 'Title', STATUS )

*  End the NDF context.
      CALL NDF_END( STATUS )

*  If an error occurred, then report context information.
      IF ( STATUS .NE. SAI__OK ) THEN
         CALL ERR_REP( 'ADD_ERR',
     :   'ADD: Error adding two NDF data structures.', STATUS )
      END IF

      END
```

The following is an example ADAM interface file (add.ifl) for the application above.

```
   interface ADD

      parameter IN1                 # First input NDF
         position 1
         prompt   'First input NDF'
      endparameter

      parameter IN2                 # Second input NDF
         position 2
         prompt   'Second input NDF'
      endparameter

      parameter OUT                 # Output NDF
         position 3
         prompt   'Output NDF'
      endparameter

      parameter TITLE               # Title for output NDF
         type     'LITERAL'
         prompt   'Title for output NDF'
         vpath    'DEFAULT'
         default  !
      endparameter

   endinterface
```

A.8 NDFTRACE — Trace an NDF Structure

The following rather long example is an application to display the attributes of an NDF data structure. It is probably not typical of the use to which the NDF_ routines will be put, but it demonstrates the use of most of the enquiry routines and provides a "guided tour" of the NDF components.

```
      SUBROUTINE NDFTRACE( STATUS )
*+
*  Name:
*     NDFTRACE

*  Purpose:
*     Display the attributes of an NDF data structure.

*  Description:
*     This routine displays the attributes of an NDF data structure
*     including its name, the values of its character components, its
*     shape and the attributes of its data array and of any other array
*     components present. A list of any extensions present, together
*     with their HDS data types, is also included.

*  ADAM Parameters:
*     NDF = NDF (Read)
*        The NDF data structure whose attributes are to be displayed.

*-

*  Type Definitions:
      IMPLICIT NONE              ! No implicit typing

*  Global Constants:
      INCLUDE 'SAE_PAR'          ! Standard SAE constants
      INCLUDE 'DAT_PAR'          ! DAT_ public constants
      INCLUDE 'NDF_PAR'          ! NDF_ public constants
      INCLUDE 'PRM_PAR'          ! PRIMDAT primitive data constants

*  Status:
      INTEGER STATUS             ! Global status

*  Local Variables:
      BYTE BADBIT                ! Bad-bits mask
      CHARACTER * ( 35 ) APPN    ! Last recorded application name
      CHARACTER * ( 8 ) BINSTR   ! Binary bad-bits mask string
      CHARACTER * ( DAT__SZLOC ) XLOC ! Extension locator
      CHARACTER * ( DAT__SZTYP ) TYPE ! Extension type
      CHARACTER * ( NDF__MXDIM * ( 2 * VAL__SZI + 3 ) - 2 ) BUF
                                 ! Text buffer for shape information
      CHARACTER * ( NDF__SZFRM ) FORM ! Storage form
      CHARACTER * ( NDF__SZFTP ) FTYPE ! Full data type
      CHARACTER * ( NDF__SZHDT ) CREAT ! History component creation date
      CHARACTER * ( NDF__SZHDT ) DATE ! Date of last history update
      CHARACTER * ( NDF__SZHUM ) HMODE ! History update mode
      CHARACTER * ( NDF__SZXNM ) XNAME ! Extension name
```

```
      INTEGER BBI                 ! Bad-bits value as an integer
      INTEGER DIGVAL              ! Binary digit value
      INTEGER DIM( NDF__MXDIM )  ! Dimension sizes
      INTEGER I                   ! Loop counter for dimensions
      INTEGER IAXIS               ! Loop counter for axes
      INTEGER IDIG                ! Loop counter for binary digits
      INTEGER INDF                ! NDF identifier
      INTEGER LBND( NDF__MXDIM ) ! Lower pixel-index bounds
      INTEGER N                   ! Loop counter for extensions
      INTEGER NC                  ! Character count
      INTEGER NDIM                ! Number of dimensions
      INTEGER NEXTN               ! Number of extensions
      INTEGER NREC                ! Number of history records
      INTEGER SIZE                ! Total number of pixels
      INTEGER UBND( NDF__MXDIM ) ! Upper pixel-index bounds
      LOGICAL BAD                 ! Bad pixel flag
      LOGICAL THERE               ! Whether NDF component is defined

*  Internal References:
      INCLUDE 'NUM_DEC_CVT'       ! NUM_ type conversion routines
      INCLUDE 'NUM_DEF_CVT'

*.

*  Check inherited global status.
      IF ( STATUS .NE. SAI__OK ) RETURN

*  Obtain an identifier for the NDF structure to be examined.
      CALL NDF_ASSOC( 'NDF', 'READ', INDF, STATUS )

*  Display the NDF's name.
      CALL MSG_BLANK( STATUS )
      CALL NDF_MSG( 'NDF', INDF )
      CALL MSG_OUT( 'HEADER', '   NDF structure ^NDF:', STATUS )

*  Character components:
*  ====================
*  See if the title component is defined. If so, then display its
*  value.
      CALL NDF_STATE( INDF, 'Title', THERE, STATUS )
      IF ( THERE ) THEN
         CALL NDF_CMSG( 'TITLE', INDF, 'Title', STATUS )
         CALL MSG_OUT( 'TITLE', '      Title:  ^TITLE', STATUS )
      END IF

*  See if the label component is defined. If so, then display its
*  value.
      CALL NDF_STATE( INDF, 'Label', THERE, STATUS )
      IF ( THERE ) THEN
         CALL NDF_CMSG( 'LABEL', INDF, 'Label', STATUS )
         CALL MSG_OUT( 'LABEL', '      Label:  ^LABEL', STATUS )
      END IF

*  See if the units component is defined. If so, then display its
```

```
*  value.
      CALL NDF_STATE( INDF, 'Units', THERE, STATUS )
      IF ( THERE ) THEN
         CALL NDF_CMSG( 'UNITS', INDF, 'Units', STATUS )
         CALL MSG_OUT( 'UNITS', '      Units:  ^UNITS', STATUS )
      END IF

*  NDF shape:
*  =========
*  Obtain the dimension sizes.
      CALL NDF_DIM( INDF, NDF__MXDIM, DIM, NDIM, STATUS )

*  Display a header for this information.
      CALL MSG_BLANK( STATUS )
      CALL MSG_OUT( 'SHAPE_HEADER', '   Shape:', STATUS )

*  Display the number of dimensions.
      CALL MSG_SETI( 'NDIM', NDIM )
      CALL MSG_OUT( 'DIMENSIONALITY',
     : '      No. of dimensions:  ^NDIM', STATUS )

*  Construct a string showing the dimension sizes.
      NC = 0
      DO 1 I = 1, NDIM
          IF ( I .GT. 1 ) CALL CHR_PUTC( ' x ', BUF, NC )
          CALL CHR_PUTI( DIM( I ), BUF, NC )
 1    CONTINUE
      CALL MSG_SETC( 'DIMS', BUF( : NC ) )

*  Display the dimension size information.
      CALL MSG_OUT( 'DIMENSIONS',
     : '      Dimension size(s):  ^DIMS', STATUS )

*  Obtain the pixel-index bounds.
      CALL NDF_BOUND( INDF, NDF__MXDIM, LBND, UBND, NDIM, STATUS )

*  Construct a string showing the pixel-index bounds.
      NC = 0
       DO 2 I = 1, NDIM
         IF ( I .GT. 1 ) CALL CHR_PUTC( ', ', BUF, NC )
         CALL CHR_PUTI( LBND( I ), BUF, NC )
         CALL CHR_PUTC( ':', BUF, NC )
         CALL CHR_PUTI( UBND( I ), BUF, NC )
 2    CONTINUE
      CALL MSG_SETC( 'BNDS', BUF( : NC ) )

*  Display the pixel-index bounds information.
      CALL MSG_OUT( 'BOUNDS',
     : '      Pixel bounds     :  ^BNDS', STATUS )

*  Obtain the NDF size and display this information.
      CALL NDF_SIZE( INDF, SIZE, STATUS )
      CALL MSG_SETI( 'SIZE', SIZE )
      CALL MSG_OUT( 'SIZE',
```

```
     : '      Total pixels     :  ^SIZE ', STATUS )

*  Axis component:
*  ==============
*  See if the axis coordinate system is defined. If so then output a header
*  for it.
      CALL NDF_STATE( INDF, 'Axis', THERE, STATUS )
      IF ( THERE ) THEN
         CALL MSG_BLANK( STATUS )
         CALL MSG_OUT( 'AXIS_HEADER', '   Axes:', STATUS )

*  Loop to obtain the label and units for each axis and display them.
         DO 3 IAXIS = 1, NDIM
            CALL MSG_SETI( 'IAXIS', IAXIS )
            CALL NDF_ACMSG( 'LABEL', INDF, 'Label', IAXIS, STATUS )
            CALL NDF_ACMSG( 'UNITS', INDF, 'Units', IAXIS, STATUS )
            CALL MSG_OUT( 'AXIS_LABEL',
     :      '      ^IAXIS: ^LABEL (^UNITS)', STATUS )
 3       CONTINUE
      END IF

*  Data component:
*  ==============
*  Obtain the data component attributes.
      CALL NDF_FTYPE( INDF, 'Data', FTYPE, STATUS )
      CALL NDF_FORM( INDF, 'Data', FORM, STATUS )

*  Display the data component attributes.
      CALL MSG_BLANK( STATUS )
      CALL MSG_OUT( 'DATA_HEADER', '   Data Component:', STATUS )
      CALL MSG_SETC( 'FTYPE', FTYPE )
      CALL MSG_OUT( 'DATA_TYPE', '      Type        :  ^FTYPE', STATUS )
      CALL MSG_SETC( 'FORM', FORM )
      CALL MSG_OUT( 'DATA_FORM', '      Storage form:  ^FORM', STATUS )

*  Determine if the data values are defined. Issue a warning message if
*  they are not.
      CALL NDF_STATE( INDF, 'Data', THERE, STATUS )
      IF ( .NOT. THERE ) THEN
         CALL MSG_OUT( 'DATA_UNDEF',
     :   '      WARNING: the Data component values are not defined',
     :                 STATUS )

*  Disable automatic quality masking and see if the data component may
*  contain bad pixels. If so, then display an appropriate message.
      ELSE
         CALL NDF_SQMF( .FALSE., INDF, STATUS )
         CALL NDF_BAD( INDF, 'Data', .FALSE., BAD, STATUS )
         IF ( BAD ) THEN
            CALL MSG_OUT( 'DATA_ISBAD',
     :      '      Bad pixels may be present', STATUS )

*  If there were no bad pixels present, then re-enable quality masking
*  and test again. Issue an appropriate message.
```

```
            ELSE
               CALL NDF_SQMF( .TRUE., INDF, STATUS )
               CALL NDF_BAD( INDF, 'Data', .FALSE., BAD, STATUS )
               IF ( .NOT. BAD ) THEN
                  CALL MSG_OUT( 'DATA_NOBAD',
     :            '      There are no bad pixels present', STATUS )
               ELSE
                  CALL MSG_OUT( 'DATA_QBAD',
     :            '      Bad pixels may be introduced via the Quality ' //
     :            'component', STATUS )
               END IF
            END IF
         END IF

*  Variance component:
*  ==================
*  See if the variance component is defined.  If so, then obtain its
*  attributes.
         CALL NDF_STATE( INDF, 'Variance', THERE, STATUS )
         IF ( THERE ) THEN
            CALL NDF_FTYPE( INDF, 'Variance', FTYPE, STATUS )
            CALL NDF_FORM( INDF, 'Variance', FORM, STATUS )

*  Display the variance component attributes.
            CALL MSG_BLANK( STATUS )
            CALL MSG_OUT( 'VAR_HEADER', '   Variance Component:', STATUS )
            CALL MSG_SETC( 'FTYPE', FTYPE )
            CALL MSG_OUT( 'VAR_TYPE', '      Type        :  ^FTYPE',
     :                    STATUS )
            CALL MSG_SETC( 'FORM', FORM )
            CALL MSG_OUT( 'VAR_FORM', '      Storage form:  ^FORM',
     :                    STATUS )

*  Disable automatic quality masking and see if the variance component
*  may contain bad pixels. If so, then display an appropriate message.
            CALL NDF_SQMF( .FALSE., INDF, STATUS )
            CALL NDF_BAD( INDF, 'Variance', .FALSE., BAD, STATUS )
            IF ( BAD ) THEN
               CALL MSG_OUT( 'VAR_ISBAD',
     :         '      Bad pixels may be present', STATUS )

*  If there were no bad pixels present, then re-enable quality masking
*  and test again. Issue an appropriate message.
            ELSE
               CALL NDF_SQMF( .TRUE., INDF, STATUS )
               CALL NDF_BAD( INDF, 'Variance', .FALSE., BAD, STATUS )
               IF ( .NOT. BAD ) THEN
                  CALL MSG_OUT( 'VAR_NOBAD',
     :            '      There are no bad pixels present', STATUS )
               ELSE
                  CALL MSG_OUT( 'VAR_QBAD',
     :            '      Bad pixels may be introduced via the Quality ' //
     :            'component', STATUS )
               END IF
```

```
         END IF
      END IF

*  Quality component:
*  =================
*  See if the quality component is defined. If so, then obtain its
*  attributes.
      CALL NDF_STATE( INDF, 'Quality', THERE, STATUS )
      IF ( THERE ) THEN
         CALL NDF_FORM( INDF, 'Quality', FORM, STATUS )

*  Display the quality component attributes.
         CALL MSG_BLANK( STATUS )
         CALL MSG_OUT( 'QUALITY_HEADER', '   Quality Component:',
     :                 STATUS )
         CALL MSG_SETC( 'FORM', FORM )
         CALL MSG_OUT( 'QUALITY_FORM', '      Storage form :  ^FORM',
     :                  STATUS )

*  Obtain the bad-bits mask value.
         CALL NDF_BB( INDF, BADBIT, STATUS )

*  Generate a binary representation in a character string.
         BBI = NUM_UBTOI( BADBIT )
         DIGVAL = 2 ** 7
         DO 4 IDIG = 1, 8
            IF ( BBI .GE. DIGVAL ) THEN
               BINSTR( IDIG : IDIG ) = '1'
               BBI = BBI - DIGVAL
            ELSE
               BINSTR( IDIG : IDIG ) = '0'
            END IF
            DIGVAL = DIGVAL / 2
 4       CONTINUE

*  Display the bad-bits mask information.
         CALL MSG_SETI( 'BADBIT', NUM_UBTOI( BADBIT ) )
         CALL MSG_SETC( 'BINARY', BINSTR )
         CALL MSG_OUT( 'QUALITY_BADBIT',
     :   '      Bad-bits mask:  ^BADBIT (binary ^BINARY)', STATUS )
      END IF

*  Extensions:
*  ==========
*  Determine how many extensions are present.
      CALL NDF_XNUMB( INDF, NEXTN, STATUS )

*  Display a heading for the extensions.
      IF ( NEXTN .GT. 0 ) THEN
         CALL MSG_BLANK( STATUS )
         CALL MSG_OUT( 'EXTN_HEADER', '   Extensions:', STATUS )

*  Loop to obtain the name and HDS data type of each extension.
         DO 5 N = 1, NEXTN
```

```
            CALL NDF_XNAME( INDF, N, XNAME, STATUS )
            CALL NDF_XLOC( INDF, XNAME, 'READ', XLOC, STATUS )
            CALL DAT_TYPE( XLOC, TYPE, STATUS )
            CALL DAT_ANNUL( XLOC, STATUS )

*  Display the information for each extension.
            CALL MSG_SETC( 'TYPE', TYPE )
            CALL MSG_OUT( 'EXTN',
     :      '      ' // XNAME // '  <^TYPE>', STATUS )
 5       CONTINUE
      END IF

*  History:
*  =======
*  See if a history component is present.
      CALL NDF_STATE( INDF, 'History', THERE, STATUS )

*  If so, then obtain its attributes.
      IF ( THERE ) THEN
         CALL NDF_HINFO( INDF, 'CREATED', 0, CREAT, STATUS )
         CALL NDF_HNREC( INDF, NREC, STATUS )
         CALL NDF_HINFO( INDF, 'MODE', 0,  HMODE, STATUS )
         CALL NDF_HINFO( INDF, 'DATE', NREC, DATE, STATUS )
         CALL NDF_HINFO( INDF, 'APPLICATION', NREC, APPN, STATUS )

*  Display the history component attributes.
         CALL MSG_BLANK( STATUS )
         CALL MSG_OUT( 'HISTORY_HEADER', '   History Component:',
     :                 STATUS )
         CALL MSG_SETC( 'CREAT', CREAT( : 20 ) )
         CALL MSG_OUT( 'HISTORY_CREAT',
     :                 '      Created    :  ^CREAT', STATUS )
         CALL MSG_SETI( 'NREC', NREC )
         CALL MSG_OUT( 'HISTORY_NREC',
     :                 '      No. records:  ^NREC', STATUS )
         CALL MSG_SETC( 'DATE', DATE( : 20 ) )
         CALL MSG_SETC( 'APPN', APPN )
         CALL MSG_OUT( 'HISTORY_DATE',
     :                 '      Last update:  ^DATE (^APPN)', STATUS )
         CALL MSG_SETC( 'HMODE', HMODE )
         CALL MSG_OUT( 'HISTORY_HMODE',
     :                 '      Update mode:  ^HMODE', STATUS )
      END IF
      CALL MSG_BLANK( STATUS )

*  Clean up:
*  ========
*  Annul the NDF identifier.
      CALL NDF_ANNUL( INDF, STATUS )

*  If an error occurred, then report context information.
      IF ( STATUS .NE. SAI__OK ) THEN
         CALL ERR_REP( 'NDFTRACE_ERR',
     :   'NDFTRACE: Error displaying the attributes of an NDF ' //
```

```
     :   'data structure.', STATUS )
      END IF

      END
```

The following is an example ADAM interface file (ndftrace.ifl) for the application above.

```
interface NDFTRACE

   parameter NDF                 # NDF to be inspected
      position 1
      prompt   'Data structure'
   endparameter

endinterface
```

B ALPHABETICAL LIST OF FORTRAN ROUTINES

NDF_ACGET(INDF, COMP, IAXIS, VALUE, STATUS)
Obtain the value of an NDF axis character component

NDF_ACLEN(INDF, COMP, IAXIS, LENGTH, STATUS)
Determine the length of an NDF axis character component

NDF_ACMSG(TOKEN, INDF, COMP, IAXIS, STATUS)
Assign the value of an NDF axis character component to a message token

NDF_ACPUT(VALUE, INDF, COMP, IAXIS, STATUS)
Assign a value to an NDF axis character component

NDF_ACRE(INDF, STATUS)
Ensure that an axis coordinate system exists for an NDF

NDF_AFORM(INDF, COMP, IAXIS, FORM, STATUS)
Obtain the storage form of an NDF axis array

NDF_AMAP(INDF, COMP, IAXIS, TYPE, MMOD, PNTR, EL, STATUS)
Obtain mapped access to an NDF axis array

NDF_ANNUL(INDF, STATUS)
Annul an NDF identifier

NDF_ANORM(INDF, IAXIS, NORM, STATUS)
Obtain the logical value of an NDF axis normalisation flag

NDF_AREST(INDF, COMP, IAXIS, STATUS)
Reset an NDF axis component to an undefined state

NDF_ASNRM(NORM, INDF, IAXIS, STATUS)
Set a new value for an NDF axis normalisation flag

NDF_ASSOC(PARAM, MODE, INDF, STATUS)
Associate an existing NDF with an ADAM parameter

NDF_ASTAT(INDF, COMP, IAXIS, STATE, STATUS)
Determine the state of an NDF axis component (defined or undefined)

NDF_ASTYP(TYPE, INDF, COMP, IAXIS, STATUS)
Set a new numeric type for an NDF axis array

NDF_ATYPE(INDF, COMP, IAXIS, TYPE, STATUS)
Obtain the numeric type of an NDF axis array

NDF_AUNMP(INDF, COMP, IAXIS, STATUS)
Unmap an NDF axis array component

NDF_BAD(INDF, COMP, CHECK, BAD, STATUS)
Determine if an NDF array component may contain bad pixels

NDF_BASE(INDF1, INDF2, STATUS)
Obtain an identifier for a base NDF

NDF_BB(INDF, BADBIT, STATUS)
Obtain the bad-bits mask value for the quality component of an NDF

NDF_BEGIN
Begin a new NDF context

NDF_BLOCK(INDF1, NDIM, MXDIM, IBLOCK, INDF2, STATUS)
Obtain an NDF section containing a block of adjacent pixels

NDF_BOUND(INDF, NDIMX, LBND, UBND, NDIM, STATUS)
Enquire the pixel-index bounds of an NDF

NDF_CGET(INDF, COMP, VALUE, STATUS)
Obtain the value of an NDF character component

NDF_CHUNK(INDF1, MXPIX, ICHUNK, INDF2, STATUS)
Obtain an NDF section containing a chunk of contiguous pixels

NDF_CINP(PARAM, INDF, COMP, STATUS)
Obtain an NDF character component value via the ADAM parameter system

NDF_CLEN(INDF, COMP, LENGTH, STATUS)
Determine the length of an NDF character component

NDF_CLONE(INDF1, INDF2, STATUS)
Clone an NDF identifier

NDF_CMPLX(INDF, COMP, CMPLX, STATUS)
Determine whether an NDF array component holds complex values

NDF_CMSG(TOKEN, INDF, COMP, STATUS)
Assign the value of an NDF character component to a message token

NDF_COPY(INDF1, PLACE, INDF2, STATUS)
Copy an NDF to a new location

NDF_CPUT(VALUE, INDF, COMP, STATUS)
Assign a value to an NDF character component

NDF_CREAT(PARAM, FTYPE, NDIM, LBND, UBND, INDF, STATUS)
Create a new simple NDF via the ADAM parameter system

NDF_CREP(PARAM, FTYPE, NDIM, UBND, INDF, STATUS)
Create a new primitive NDF via the ADAM parameter system

NDF_CREPL(PARAM, PLACE, STATUS)
Create a new NDF placeholder via the ADAM parameter system

NDF_DELET(INDF, STATUS)
Delete an NDF

NDF_DIM(INDF, NDIMX, DIM, NDIM, STATUS)
Enquire the dimension sizes of an NDF

NDF_END(STATUS)
End the current NDF context

NDF_EXIST(PARAM, MODE, INDF, STATUS)
See if an existing NDF is associated with an ADAM parameter.

NDF_FIND(LOC, NAME, INDF, STATUS)
Find an NDF in an HDS structure and import it into the NDF_ system

NDF_FORM(INDF, COMP, FORM, STATUS)
Obtain the storage form of an NDF array component

NDF_FTYPE(INDF, COMP, FTYPE, STATUS)
Obtain the full type of an NDF array component

NDF_GTDLT(INDF, COMP, ZAXIS, ZTYPE, ZRATIO, STATUS)
Get compression details for a DELTA compressed NDF array component

NDF_GTSZx(INDF, COMP, SCALE, ZERO, STATUS)
Get scale and zero factors for a scaled array in an NDF

NDF_GTUNE(TPAR, VALUE, STATUS)
Obtain the value of an NDF_ system tuning parameter

NDF_GTWCS(INDF, IWCS, STATUS)
Obtain world coordinate system information from an NDF

NDF_HAPPN(APPN, STATUS)
Declare a new application name for NDF history recording

NDF_HCOPY(INDF1, INDF2, STATUS)
Copy history information from one NDF to another

NDF_HCRE(INDF, STATUS)
Ensure that a history component exists for an NDF

NDF_HDEF(INDF, APPN, STATUS)
Write default history information to an NDF

NDF_HECHO(NLINES, TEXT, STATUS)
Write out lines of history text

NDF_HEND(STATUS)
End NDF history recording for the current application

NDF_HFIND(INDF, YMDHM, SEC, EQ, IREC, STATUS)
Find an NDF history record by date and time

NDF_HGMOD(INDF, HMODE, STATUS)
Get the history update mode for an NDF

NDF_HINFO(INDF, ITEM, IREC, VALUE, STATUS)
Obtain information about an NDF's history component

NDF_HNREC(INDF, NREC, STATUS)
Determine the number of NDF history records present

NDF_HOUT(INDF, IREC, ROUTIN, STATUS)
Display text from an NDF history record

NDF_HPURG(INDF, IREC1, IREC2, STATUS)
Delete a range of records from an NDF history component

NDF_HPUT(HMODE, APPN, REPL, NLINES, TEXT, TRANS, WRAP, RJUST, INDF, STATUS)
Write history information to an NDF

NDF_HSMOD(HMODE, INDF, STATUS)
Set the history update mode for an NDF

NDF_ISACC(INDF, ACCESS, ISACC, STATUS)
Determine whether a specified type of NDF access is available

NDF_ISBAS(INDF, ISBAS, STATUS)
Enquire if an NDF is a base NDF

NDF_ISIN(INDF1, INDF2, ISIN, STATUS)
Enquire if one NDF is contained within another NDF

NDF_ISTMP(INDF, ISTMP, STATUS)
Determine if an NDF is temporary

NDF_LOC(INDF, MODE, LOC, STATUS)
Obtain an HDS locator for an NDF

NDF_MAP(INDF, COMP, TYPE, MMOD, PNTR, EL, STATUS)
Obtain mapped access to an array component of an NDF

NDF_MAPQL(INDF, PNTR, EL, BAD, STATUS)
Map the quality component of an NDF as an array of logical values

NDF_MAPZ(INDF, COMP, TYPE, MMOD, RPNTR, IPNTR, EL, STATUS)
Obtain complex mapped access to an array component of an NDF

NDF_MBAD(BADOK, INDF1, INDF2, COMP, CHECK, BAD, STATUS)
Merge the bad-pixel flags of the array components of a pair of NDFs

NDF_MBADN(BADOK, N, NDFS, COMP, CHECK, BAD, STATUS)
Merge the bad-pixel flags of the array components of a number of NDFs

NDF_MBND(OPTION, INDF1, INDF2, STATUS)
Match the pixel-index bounds of a pair of NDFs

NDF_MBNDN(OPTION, N, NDFS, STATUS)
Match the pixel-index bounds of a number of NDFs

NDF_MSG(TOKEN, INDF)
Assign the name of an NDF to a message token

NDF_MTYPE(TYPLST, INDF1, INDF2, COMP, ITYPE, DTYPE, STATUS)
Match the types of the array components of a pair of NDFs

NDF_MTYPN(TYPLST, N, NDFS, COMP, ITYPE, DTYPE, STATUS)
Match the types of the array components of a number of NDFs

NDF_NBLOC(INDF, NDIM, MXDIM, NBLOCK, STATUS)
Determine the number of blocks of adjacent pixels in an NDF

NDF_NCHNK(INDF, MXPIX, NCHUNK, STATUS)
Determine the number of chunks of contiguous pixels in an NDF

NDF_NEW(FTYPE, NDIM, LBND, UBND, PLACE, INDF, STATUS)
Create a new simple NDF

NDF_NEWP(FTYPE, NDIM, UBND, PLACE, INDF, STATUS)
Create a new primitive NDF

NDF_NOACC(ACCESS, INDF, STATUS)
Disable a specified type of access to an NDF

NDF_OPEN(LOC, NAME, MODE, STAT, INDF, PLACE, STATUS)
Open an existing or new NDF

NDF_PLACE(LOC, NAME, PLACE, STATUS)
Obtain an NDF placeholder

NDF_PROP(INDF1, CLIST, PARAM, INDF2, STATUS)
Propagate NDF information to create a new NDF via the ADAM parameter system

NDF_PTSZ*x*(SCALE, ZERO, INDF, COMP, STATUS)
Store new scale and zero factors for a scaled array in an NDF

NDF_PTWCS(IWCS, INDF, STATUS)
Store world coordinate system information in an NDF

NDF_QMASK(QUAL, BADBIT)
Combine an NDF quality value with a bad-bits mask to give a logical result

NDF_QMF(INDF, QMF, STATUS)
Obtain the value of an NDF's quality masking flag

NDF_RESET(INDF, COMP, STATUS)
Reset an NDF component to an undefined state

NDF_SAME(INDF1, INDF2, SAME, ISECT, STATUS)
Enquire if two NDFs are part of the same base NDF

NDF_SBAD(BAD, INDF, COMP, STATUS)
Set the bad-pixel flag for an NDF array component

NDF_SBB(BADBIT, INDF, STATUS)
Set a bad-bits mask value for the quality component of an NDF

NDF_SBND(NDIM, LBND, UBND, INDF, STATUS)
Set new pixel-index bounds for an NDF

NDF_SCOPY(INDF1, CLIST, PLACE, INDF2, STATUS)
Selectively copy NDF components to a new location

NDF_SCTYP(INDF, COMP, TYPE, STATUS)
Obtain the numeric type of a scaled NDF array component

NDF_SECT(INDF1, NDIM, LBND, UBND, INDF2, STATUS)
Create an NDF section

NDF_SHIFT(NSHIFT, SHIFT, INDF, STATUS)
Apply pixel-index shifts to an NDF

NDF_SIZE(INDF, NPIX, STATUS)
Determine the size of an NDF

NDF_SQMF(QMF, INDF, STATUS)
Set a new logical value for an NDF's quality masking flag

NDF_SSARY(IARY1, INDF, IARY2, STATUS)
Create an array section, using an NDF section as a template

NDF_STATE(INDF, COMP, STATE, STATUS)
Determine the state of an NDF component (defined or undefined)

NDF_STYPE(FTYPE, INDF, COMP, STATUS)
Set a new type for an NDF array component

NDF_TEMP(PLACE, STATUS)
Obtain a placeholder for a temporary NDF

NDF_TUNE(VALUE, TPAR, STATUS)
Set an NDF_ system tuning parameter

NDF_TYPE(INDF, COMP, TYPE, STATUS)
Obtain the numeric type of an NDF array component

NDF_UNMAP(INDF, COMP, STATUS)
Unmap an NDF or a mapped NDF array

NDF_VALID(INDF, VALID, STATUS)
Determine whether an NDF identifier is valid

NDF_XDEL(INDF, XNAME, STATUS)
Delete a specified NDF extension

NDF_XGT0x(INDF, XNAME, CMPT, VALUE, STATUS)
Read a scalar value from a component within a named NDF extension

NDF_XIARY(INDF, XNAME, CMPT, MODE, IARY, STATUS)
Obtain access to an array stored in an NDF extension

NDF_XLOC(INDF, XNAME, MODE, LOC, STATUS)
Obtain access to a named NDF extension via an HDS locator

NDF_XNAME(INDF, N, XNAME, STATUS)
Obtain the name of the N'th extension in an NDF

NDF_XNEW(INDF, XNAME, TYPE, NDIM, DIM, LOC, STATUS)
Create a new extension in an NDF

NDF_XNUMB(INDF, NEXTN, STATUS)
Determine the number of extensions in an NDF

NDF_XPT0x(VALUE, INDF, XNAME, CMPT, STATUS)
Write a scalar value to a component within a named NDF extension

NDF_XSTAT(INDF, XNAME, THERE, STATUS)
Determine if a named NDF extension exists

NDF_ZDELT(INDF1, COMP, MINRAT, ZAXIS, TYPE, PLACE, INDF2, ZRATIO, STATUS)
Create a compressed copy of an NDF using DELTA compression

NDF_ZSCAL(INDF1, TYPE, SCALE, ZERO, PLACE, INDF2, STATUS)
Create a compressed copy of an NDF using SCALE compression

C CLASSIFIED LIST OF FORTRAN ROUTINES

C.1 Access to Existing NDFs

NDF_ASSOC(PARAM, MODE, INDF, STATUS)
Associate an existing NDF with an ADAM parameter

NDF_EXIST(PARAM, MODE, INDF, STATUS)
See if an existing NDF is associated with an ADAM parameter.

NDF_FIND(LOC, NAME, INDF, STATUS)
Find an NDF in an HDS structure and import it into the NDF_ system

NDF_OPEN(LOC, NAME, MODE, STAT, INDF, PLACE, STATUS)
Open an existing or new NDF

C.2 Enquiring NDF Attributes

NDF_BOUND(INDF, NDIMX, LBND, UBND, NDIM, STATUS)
Enquire the pixel-index bounds of an NDF

NDF_DIM(INDF, NDIMX, DIM, NDIM, STATUS)
Enquire the dimension sizes of an NDF

NDF_ISACC(INDF, ACCESS, ISACC, STATUS)
Determine whether a specified type of NDF access is available

NDF_ISBAS(INDF, ISBAS, STATUS)
Enquire if an NDF is a base NDF

NDF_ISIN(INDF1, INDF2, ISIN, STATUS)
Enquire if one NDF is contained within another NDF

NDF_ISTMP(INDF, ISTMP, STATUS)
Determine if an NDF is temporary

NDF_NBLOC(INDF, NDIM, MXDIM, NBLOCK, STATUS)
Determine the number of blocks of adjacent pixels in an NDF

NDF_NCHNK(INDF, MXPIX, NCHUNK, STATUS)
Determine the number of chunks of contiguous pixels in an NDF

NDF_SAME(INDF1, INDF2, SAME, ISECT, STATUS)
Enquire if two NDFs are part of the same base NDF

NDF_SIZE(INDF, NPIX, STATUS)
Determine the size of an NDF

NDF_VALID(INDF, VALID, STATUS)
Determine whether an NDF identifier is valid

C.3 Enquiring Component Attributes

NDF_BAD(INDF, COMP, CHECK, BAD, STATUS)
Determine if an NDF array component may contain bad pixels

NDF_BB(INDF, BADBIT, STATUS)
Obtain the bad-bits mask value for the quality component of an NDF

NDF_CLEN(INDF, COMP, LENGTH, STATUS)
Determine the length of an NDF character component

NDF_CMPLX(INDF, COMP, CMPLX, STATUS)
Determine whether an NDF array component holds complex values

NDF_FORM(INDF, COMP, FORM, STATUS)
Obtain the storage form of an NDF array component

NDF_FTYPE(INDF, COMP, FTYPE, STATUS)
Obtain the full type of an NDF array component

NDF_QMF(INDF, QMF, STATUS)
Obtain the value of an NDF's quality masking flag

NDF_STATE(INDF, COMP, STATE, STATUS)
Determine the state of an NDF component (defined or undefined)

NDF_TYPE(INDF, COMP, TYPE, STATUS)
Obtain the numeric type of an NDF array component

C.4 Creating and Deleting NDFs

NDF_CREAT(PARAM, FTYPE, NDIM, LBND, UBND, INDF, STATUS)
Create a new simple NDF via the ADAM parameter system

NDF_CREP(PARAM, FTYPE, NDIM, UBND, INDF, STATUS)
Create a new primitive NDF via the ADAM parameter system

NDF_DELET(INDF, STATUS)
Delete an NDF

NDF_NEW(FTYPE, NDIM, LBND, UBND, PLACE, INDF, STATUS)
Create a new simple NDF

NDF_NEWP(FTYPE, NDIM, UBND, PLACE, INDF, STATUS)
Create a new primitive NDF

NDF_OPEN(LOC, NAME, MODE, STAT, INDF, PLACE, STATUS)
Open an existing or new NDF

NDF_PROP(INDF1, CLIST, PARAM, INDF2, STATUS)
Propagate NDF information to create a new NDF via the ADAM parameter system

NDF_SCOPY(INDF1, CLIST, PLACE, INDF2, STATUS)
Selectively copy NDF components to a new location

C.5 Setting NDF Attributes

NDF_NOACC(ACCESS, INDF, STATUS)
Disable a specified type of access to an NDF

NDF_SBND(NDIM, LBND, UBND, INDF, STATUS)
Set new pixel-index bounds for an NDF

NDF_SHIFT(NSHIFT, SHIFT, INDF, STATUS)
Apply pixel-index shifts to an NDF

C.6 Setting Component Attributes

NDF_RESET(INDF, COMP, STATUS)
Reset an NDF component to an undefined state

NDF_SBAD(BAD, INDF, COMP, STATUS)
Set the bad-pixel flag for an NDF array component

NDF_SBB(BADBIT, INDF, STATUS)
Set a bad-bits mask value for the quality component of an NDF

NDF_SQMF(QMF, INDF, STATUS)
Set a new logical value for an NDF's quality masking flag

NDF_STYPE(FTYPE, INDF, COMP, STATUS)
Set a new type for an NDF array component

C.7 Access to Component Values

NDF_CGET(INDF, COMP, VALUE, STATUS)
Obtain the value of an NDF character component

NDF_CPUT(VALUE, INDF, COMP, STATUS)
Assign a value to an NDF character component

NDF_MAP(INDF, COMP, TYPE, MMOD, PNTR, EL, STATUS)
Obtain mapped access to an array component of an NDF

NDF_MAPQL(INDF, PNTR, EL, BAD, STATUS)
Map the quality component of an NDF as an array of logical values

NDF_MAPZ(INDF, COMP, TYPE, MMOD, RPNTR, IPNTR, EL, STATUS)
Obtain complex mapped access to an array component of an NDF

NDF_QMASK(QUAL, BADBIT)
Combine an NDF quality value with a bad-bits mask to give a logical result

NDF_UNMAP(INDF, COMP, STATUS)
Unmap an NDF or a mapped NDF array

C.8 Enquiring and Setting Axis Attributes

NDF_ACLEN(INDF, COMP, IAXIS, LENGTH, STATUS)
Determine the length of an NDF axis character component

NDF_ACRE(INDF, STATUS)
Ensure that an axis coordinate system exists for an NDF

NDF_AFORM(INDF, COMP, IAXIS, FORM, STATUS)
Obtain the storage form of an NDF axis array

NDF_ANORM(INDF, IAXIS, NORM, STATUS)
Obtain the logical value of an NDF axis normalisation flag

NDF_AREST(INDF, COMP, IAXIS, STATUS)
Reset an NDF axis component to an undefined state

NDF_ASNRM(NORM, INDF, IAXIS, STATUS)
Set a new value for an NDF axis normalisation flag

NDF_ASTAT(INDF, COMP, IAXIS, STATE, STATUS)
Determine the state of an NDF axis component (defined or undefined)

NDF_ASTYP(TYPE, INDF, COMP, IAXIS, STATUS)
Set a new numeric type for an NDF axis array

NDF_ATYPE(INDF, COMP, IAXIS, TYPE, STATUS)
Obtain the numeric type of an NDF axis array

C.9 Access to Axis Values

NDF_ACGET(INDF, COMP, IAXIS, VALUE, STATUS)
Obtain the value of an NDF axis character component

NDF_ACMSG(TOKEN, INDF, COMP, IAXIS, STATUS)
Assign the value of an NDF axis character component to a message token

NDF_ACPUT(VALUE, INDF, COMP, IAXIS, STATUS)
Assign a value to an NDF axis character component

NDF_AMAP(INDF, COMP, IAXIS, TYPE, MMOD, PNTR, EL, STATUS)
Obtain mapped access to an NDF axis array

NDF_AUNMP(INDF, COMP, IAXIS, STATUS)
Unmap an NDF axis array component

C.10 Access to World Coordinate System Information

NDF_GTWCS(INDF, IWCS, STATUS)
Obtain world coordinate system information from an NDF

NDF_PTWCS(IWCS, INDF, STATUS)
Store world coordinate system information in an NDF

C.11 Creation and Control of Identifiers

NDF_ANNUL(INDF, STATUS)
Annul an NDF identifier

NDF_BASE(INDF1, INDF2, STATUS)
Obtain an identifier for a base NDF

NDF_BEGIN
Begin a new NDF context

NDF_CLONE(INDF1, INDF2, STATUS)
Clone an NDF identifier

NDF_END(STATUS)
End the current NDF context

NDF_VALID(INDF, VALID, STATUS)
Determine whether an NDF identifier is valid

C.12 Handling NDF (and Array) Sections

NDF_BASE(INDF1, INDF2, STATUS)
Obtain an identifier for a base NDF

NDF_BLOCK(INDF1, NDIM, MXDIM, IBLOCK, INDF2, STATUS)
Obtain an NDF section containing a block of adjacent pixels

NDF_CHUNK(INDF1, MXPIX, ICHUNK, INDF2, STATUS)
Obtain an NDF section containing a chunk of contiguous pixels

NDF_NBLOC(INDF, NDIM, MXDIM, NBLOCK, STATUS)
Determine the number of blocks of adjacent pixels in an NDF

NDF_NCHNK(INDF, MXPIX, NCHUNK, STATUS)
Determine the number of chunks of contiguous pixels in an NDF

NDF_SECT(INDF1, NDIM, LBND, UBND, INDF2, STATUS)
Create an NDF section

NDF_SSARY(IARY1, INDF, IARY2, STATUS)
Create an array section, using an NDF section as a template

NDF_XIARY(INDF, XNAME, CMPT, MODE, IARY, STATUS)
Obtain access to an array stored in an NDF extension

C.13 Matching and Merging Attributes

NDF_MBAD(BADOK, INDF1, INDF2, COMP, CHECK, BAD, STATUS)
Merge the bad-pixel flags of the array components of a pair of NDFs

NDF_MBADN(BADOK, N, NDFS, COMP, CHECK, BAD, STATUS)
Merge the bad-pixel flags of the array components of a number of NDFs

NDF_MBND(OPTION, INDF1, INDF2, STATUS)
Match the pixel-index bounds of a pair of NDFs

NDF_MBNDN(OPTION, N, NDFS, STATUS)
Match the pixel-index bounds of a number of NDFs

NDF_MTYPE(TYPLST, INDF1, INDF2, COMP, ITYPE, DTYPE, STATUS)
Match the types of the array components of a pair of NDFs

NDF_MTYPN(TYPLST, N, NDFS, COMP, ITYPE, DTYPE, STATUS)
Match the types of the array components of a number of NDFs

C.14 Parameter System Routines

NDF_ASSOC(PARAM, MODE, INDF, STATUS)
Associate an existing NDF with an ADAM parameter

NDF_CANCL(PARAM, STATUS)
Cancel the association of an NDF with an ADAM parameter

NDF_CINP(PARAM, INDF, COMP, STATUS)
Obtain an NDF character component value via the ADAM parameter system

NDF_CREAT(PARAM, FTYPE, NDIM, LBND, UBND, INDF, STATUS)
Create a new simple NDF via the ADAM parameter system

NDF_CREP(PARAM, FTYPE, NDIM, UBND, INDF, STATUS)
Create a new primitive NDF via the ADAM parameter system

NDF_CREPL(PARAM, PLACE, STATUS)
Create a new NDF placeholder via the ADAM parameter system

NDF_EXIST(PARAM, MODE, INDF, STATUS)
See if an existing NDF is associated with an ADAM parameter.

NDF_PROP(INDF1, CLIST, PARAM, INDF2, STATUS)
Propagate NDF information to create a new NDF via the ADAM parameter system

C.15 Message System Routines

NDF_CMSG(TOKEN, INDF, COMP, STATUS)
Assign the value of an NDF character component to a message token

NDF_MSG(TOKEN, INDF)
Assign the name of an NDF to a message token

C.16 Creating Placeholders

NDF_CREPL(PARAM, PLACE, STATUS)
Create a new NDF placeholder via the ADAM parameter system

NDF_OPEN(LOC, NAME, MODE, STAT, INDF, PLACE, STATUS)
Open an existing or new NDF

NDF_PLACE(LOC, NAME, PLACE, STATUS)
Obtain an NDF placeholder

NDF_TEMP(PLACE, STATUS)
Obtain a placeholder for a temporary NDF

C.17 Copying NDFs

NDF_COPY(INDF1, PLACE, INDF2, STATUS)
Copy an NDF to a new location

NDF_PROP(INDF1, CLIST, PARAM, INDF2, STATUS)
Propagate NDF information to create a new NDF via the ADAM parameter system

NDF_SCOPY(INDF1, CLIST, PLACE, INDF2, STATUS)
Selectively copy NDF components to a new location

C.18 Handling Extensions

NDF_XDEL(INDF, XNAME, STATUS)
Delete a specified NDF extension

NDF_XGT0x(INDF, XNAME, CMPT, VALUE, STATUS)
Read a scalar value from a component within a named NDF extension

NDF_XIARY(INDF, XNAME, CMPT, MODE, IARY, STATUS)
Obtain access to an array stored in an NDF extension

NDF_XLOC(INDF, XNAME, MODE, LOC, STATUS)
Obtain access to a named NDF extension via an HDS locator

NDF_XNAME(INDF, N, XNAME, STATUS)
Obtain the name of the N'th extension in an NDF

NDF_XNEW(INDF, XNAME, TYPE, NDIM, DIM, LOC, STATUS)
Create a new extension in an NDF

NDF_XNUMB(INDF, NEXTN, STATUS)
Determine the number of extensions in an NDF

NDF_XPT0x(VALUE, INDF, XNAME, CMPT, STATUS)
Write a scalar value to a component within a named NDF extension

NDF_XSTAT(INDF, XNAME, THERE, STATUS)
Determine if a named NDF extension exists

C.19 Handling History Information

NDF_HAPPN(APPN, STATUS)
Declare a new application name for NDF history recording

NDF_HCOPY(INDF1, INDF2, STATUS)
Copy history information from one NDF to another

NDF_HCRE(INDF, STATUS)
Ensure that a history component exists for an NDF

NDF_HDEF(INDF, APPN, STATUS)
Write default history information to an NDF

NDF_HECHO(NLINES, TEXT, STATUS)
Write out lines of history text

NDF_HEND(STATUS)
End NDF history recording for the current application

NDF_HFIND(INDF, YMDHM, SEC, EQ, IREC, STATUS)
Find an NDF history record by date and time

NDF_HGMOD(INDF, HMODE, STATUS)
Get the history update mode for an NDF

NDF_HINFO(INDF, ITEM, IREC, VALUE, STATUS)
Obtain information about an NDF's history component

NDF_HNREC(INDF, NREC, STATUS)
Determine the number of NDF history records present

NDF_HOUT(INDF, IREC, ROUTIN, STATUS)
Display text from an NDF history record

NDF_HPURG(INDF, IREC1, IREC2, STATUS)
Delete a range of records from an NDF history component

NDF_HPUT(HMODE, APPN, REPL, NLINES, TEXT, TRANS, WRAP, RJUST, INDF, STATUS)
Write history information to an NDF

NDF_HSDAT(DATE, INDF, STATUS)
Set the date and time for the next history record in an NDF

NDF_HSMOD(HMODE, INDF, STATUS)
Set the history update mode for an NDF

C.20 Tuning the NDF_ system

NDF_GTUNE(TPAR, VALUE, STATUS)
Obtain the value of an NDF_ system tuning parameter

NDF_TUNE(VALUE, TPAR, STATUS)
Set an NDF_ system tuning parameter

C.21 Compression

NDF_FORM(INDF, COMP, FORM, STATUS)
Obtain the storage form of an NDF array component

NDF_GTDLT(INDF, COMP, ZAXIS, ZTYPE, ZRATIO, STATUS)
Get compression details for a DELTA compressed NDF array component

NDF_GTSZx(INDF, COMP, SCALE, ZERO, STATUS)
Get scale and zero factors for a scaled array in an NDF

NDF_PTSZx(SCALE, ZERO, INDF, COMP, STATUS)
Store new scale and zero factors for a scaled array in an NDF

NDF_ZDELT(INDF1, COMP, MINRAT, ZAXIS, TYPE, PLACE, INDF2, ZRATIO, STATUS)
Create a compressed copy of an NDF using DELTA compression

NDF_ZSCAL(INDF1, TYPE, SCALE, ZERO, PLACE, INDF2, STATUS)
Create a compressed copy of an NDF using SCALE compression

D FORTRAN ROUTINE DESCRIPTIONS

NDF_ACGET
Obtain the value of an NDF axis character component

Description:

The routine obtains the value of the specified axis character component of an NDF (i.e. the value of the LABEL or UNITS component for an NDF axis).

Invocation:

```
CALL NDF_ACGET( INDF, COMP, IAXIS, VALUE, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the axis character component whose value is required: 'LABEL' or 'UNITS'.

IAXIS = INTEGER (Given)
Number of the axis for which a value is required.

VALUE = CHARACTER * (*) (Given and Returned)
The component's value.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If the requested axis component is in an undefined state and VALUE is set to a blank string on entry, then an appropriate default value will be returned. If VALUE is not blank on entry, then it will be returned unchanged.
- If the length of the VALUE argument is too short to accommodate the returned result without losing significant (non-blank) trailing characters, then this will be indicated by an appended ellipsis, i.e. '...'. No error will result.

NDF_ACLEN
Determine the length of an NDF axis character component

Description:

The routine returns the length of the specified axis character component of an NDF (i.e. the number of characters in the LABEL or UNITS component of an NDF axis).

Invocation:

```
CALL NDF_ACLEN( INDF, COMP, IAXIS, LENGTH, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the axis character component whose length is required: 'LABEL' or 'UNITS'.

IAXIS = INTEGER (Given)
Number of the NDF axis.

LENGTH = INTEGER (Returned)
The component's length in characters.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- The length of an NDF axis character component is normally determined by the length of the VALUE string assigned to it by a previous call to NDF_ACPUT (note that this could include trailing blanks).
- If the requested axis component is in an undefined state, then the length returned will be the number of characters in the default value which would be returned by the NDF_ACGET routine.
- A value of zero may be supplied for the IAXIS argument, in which case the routine will return the maximum component length for all the NDF axes.

NDF_ACMSG
Assign the value of an NDF axis character component to a message token

Description:

The routine assigns the value of the specified axis character component of an NDF to a message token, for use in constructing messages using the MSG_ or ERR_ routines (see SUN/104).

Invocation:

```
CALL NDF_ACMSG( TOKEN, INDF, COMP, IAXIS, STATUS )
```

Arguments:

TOKEN = CHARACTER * (*) (Given)
Name of the message token.

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the axis character component whose value is to be used: 'LABEL' or 'UNITS'.

IAXIS = INTEGER (Given)
Number of the NDF axis.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If the requested axis component is in an undefined state, then an appropriate default value will be assigned to the token.

NDF_ACPUT
Assign a value to an NDF axis character component

Description:

The routine assigns a value to the specified axis character component of an NDF (i.e. to the LABEL or UNITS component of an NDF axis).

Invocation:

```
CALL NDF_ACPUT( VALUE, INDF, COMP, IAXIS, STATUS )
```

Arguments:

VALUE = CHARACTER * (*) (Given)
The value to be assigned.

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the axis character component whose value is to be assigned: 'LABEL' or 'UNITS'.

IAXIS = INTEGER (Given)
Number of the axis to receive the new value.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- The entire VALUE string (including trailing blanks if present) is assigned to the specified axis component, whose length is adjusted to accommodate it.
- A value of zero may be given for the IAXIS argument, in which case the routine will assign the same value to all the NDF axes.
- This routine may only be used to assign values to the axes of a base NDF. If an NDF section is supplied, then it it will return without action. No error will result.

NDF_ACRE

Ensure that an axis coordinate system exists for an NDF

Description:

The routine ensures that an axis coordinate system exists for an NDF. An axis component with default coordinate values is created if necessary.

Invocation:

```
CALL NDF_ACRE( INDF, STATUS )
```

Arguments:

INDF = INTEGER (Given)

NDF identifier.

STATUS = INTEGER (Given and Returned)

The global status.

NDF_AFORM
Obtain the storage form of an NDF axis array

Description:

The routine returns the storage form of a specified NDF axis array component as an upper case character string (e.g. 'PRIMITIVE').

Invocation:

```
CALL NDF_AFORM( INDF, COMP, IAXIS, FORM, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the axis array component whose storage form is required: 'CENTRE', 'VARIANCE' or 'WIDTH'.

IAXIS = INTEGER (Given)
Number of the NDF axis for which information is required.

FORM = CHARACTER * (*) (Returned)
Storage form of the axis array.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- The symbolic constant NDF__SZFRM may be used for declaring the length of a character variable to hold the storage form of an NDF axis array. This constant is defined in the include file NDF_PAR.
- At present, the NDF_ routines only support "primitive" and "simple" arrays, so only the values 'PRIMITIVE' and 'SIMPLE' can be returned.

NDF_AMAP
Obtain mapped access to an NDF axis array

Description:

The routine obtains mapped access to an NDF axis array, returning a pointer to the mapped values and a count of the number of elements mapped.

Invocation:

```
CALL NDF_AMAP( INDF, COMP, IAXIS, TYPE, MMOD, PNTR, EL, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the axis array component to be mapped: 'CENTRE', 'VARIANCE' (or 'ERROR') or 'WIDTH'.

IAXIS = INTEGER (Given)
Number of the NDF axis whose array is to be mapped.

TYPE = CHARACTER * (*) (Given)
Numeric type to be used for access (e.g. '_REAL').

MMOD = CHARACTER * (*) (Given)
Mapping mode for access to the array: 'READ', 'UPDATE' or 'WRITE'.

PNTR(*) = INTEGER (Returned)
Pointer(s) to the mapped values (see the Notes section).

EL = INTEGER (Returned)
Number of elements mapped.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- A comma-separated list of axis array component names may also be given, in which case the routine will map all the requested axis arrays using the same numeric type and mapping mode. Pointers to the values of these mapped arrays will be returned (in the specified order) in the elements of the array PNTR, which must be of sufficient size to accommodate them.

NDF_ANNUL
Annul an NDF identifier

Description:

The routine annuls the NDF identifier supplied so that it is no longer recognised as a valid identifier by the NDF_ routines. Any resources associated with it are released and made available for re-use. If any NDF components are mapped for access, then they are automatically unmapped by this routine.

Invocation:

```
CALL NDF_ANNUL( INDF, STATUS )
```

Arguments:

INDF = INTEGER (Given and Returned)

The NDF identifier to be annulled. A value of NDF__NOID is returned (as defined in the include file NDF_PAR).

STATUS = INTEGER (Given and Returned)

The global status.

Notes:

- This routine attempts to execute even if STATUS is set on entry, although no further error report will be made if it subsequently fails under these circumstances. In particular, it will fail if the identifier supplied is not initially valid, but this will only be reported if STATUS is set to SAI__OK on entry.
- An error will result if an attempt is made to annul the last remaining identifier associated with an NDF whose DATA component has not been defined (unless it is a temporary NDF, in which case it will be deleted at this point).

NDF_ANORM
Obtain the logical value of an NDF axis normalisation flag

Description:

The routine returns a logical value for the normalisation flag associated with an NDF axis.

Invocation:

```
CALL NDF_ANORM( INDF, IAXIS, NORM, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

IAXIS = INTEGER (Given)
Number of the axis whose normalisation flag value is required.

NORM = LOGICAL (Returned)
Normalisation flag value.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- A value of zero may be supplied for the IAXIS argument, in which case the routine will return the logical "OR" of the normalisation flag values for all the NDF's axes.

NDF_AREST
Reset an NDF axis component to an undefined state

Description:

The routine resets an NDF axis component so that its value becomes undefined. It may be used to remove unwanted optional NDF axis components.

Invocation:

```
CALL NDF_AREST( INDF, COMP, IAXIS, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the axis component to be reset: 'LABEL', 'UNITS', 'VARIANCE' or 'WIDTH'.

IAXIS = INTEGER (Given)
Number of the NDF axis to be modified.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- A comma-separated list of axis component names may also be supplied, in which case each component will be reset in turn.
- A value of zero may be supplied for the IAXIS argument, in which case the same component(s) will be reset on all the NDF's axes.
- An axis component name of 'CENTRE' may not be specified for this routine because the pixel centre information cannot be reset for each axis of an NDF individually. This information may only be removed from an NDF by resetting the entire axis component. This can be done by calling the routine NDF_RESET and specifying a component name of 'AXIS'.
- This routine may only be used to reset an axis component via a base NDF. If an NDF section is supplied, then it will return without action. No error will result.
- An NDF axis array component cannot be reset while it is mapped for access, even if this is via another NDF identifier. This routine will fail, and set a STATUS value, if this is the case.

NDF_ASNRM
Set a new logical value for an NDF axis normalisation flag

Description:

The routine sets a new logical value for the normalisation flag associated with an NDF axis.

Invocation:

```
CALL NDF_ASNRM( NORM, INDF, IAXIS, STATUS )
```

Arguments:

NORM = LOGICAL (Given)
Normalisation flag value to be set.

INDF = INTEGER (Given)
NDF identifier.

IAXIS = INTEGER (Given)
Number of the NDF axis whose normalisation flag value is to be set.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- A value of zero may be supplied for the IAXIS component, in which case the routine will set the same normalisation flag value for all the NDF's axes.
- This routine may only be used to set an axis normalisation flag value for a base NDF. If an NDF section is supplied, then it will return without action. No error will result.

NDF_ASSOC
Associate an existing NDF with an ADAM parameter

Description:

The routine obtains access to an existing NDF through the ADAM parameter system, associates it with the named parameter, and issues an NDF identifier for it.

Invocation:

```
CALL NDF_ASSOC( PARAM, MODE, INDF, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)
Name of the ADAM parameter.

MODE = CHARACTER * (*) (Given)
Type of NDF access required: 'READ', 'UPDATE' or 'WRITE'.

INDF = INTEGER (Returned)
NDF identifier.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If 'WRITE' access is specified, then all the NDF's components will be reset to an undefined state ready to receive new values. If 'UPDATE' access is specified, the NDF's components will retain their values, which may then be modified.
- If this routine is called with STATUS set, then a value of NDF__NOID will be returned for the INDF argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The NDF__NOID constant is defined in the include file NDF_PAR.

NDF_ASTAT
Determine the state of an NDF axis component (defined or undefined)

Description:

The routine returns a logical value indicating whether a specified NDF axis component has a defined value (or values).

Invocation:

```
CALL NDF_ASTAT( INDF, COMP, IAXIS, STATE, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the axis component: 'CENTRE', 'LABEL', 'UNITS', 'VARIANCE' or 'WIDTH'.

IAXIS = INTEGER (Given)
Number of the NDF axis for which information is required.

STATE = LOGICAL (Returned)
Whether the specified component is defined.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- A comma-separated list of axis component names may also be given, in which case the routine will return the logical "AND" of the states of the specified components (i.e. a .TRUE. result will be returned only if all the components have defined values).
- A value of zero may be given for the IAXIS argument, in which case the routine will return the logical "AND" of the results for all the NDF's axes.

NDF_ASTYP
Set a new numeric type for an NDF axis array

Description:

The routine sets a new numeric type for an NDF axis array, causing its data storage type to be changed. If the array's values are defined, they will be converted from from the old type to the new one. If they are undefined, then no conversion will be necessary. Subsequent enquiries will reflect the new numeric type. Conversion may be performed between any numeric types supported by the NDF_ routines.

Invocation:

```
CALL NDF_ASTYP( TYPE, INDF, COMP, IAXIS, STATUS )
```

Arguments:

TYPE = CHARACTER ∗ (∗) (Given)
New numeric type for the axis array (e.g. '_DOUBLE').

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER ∗ (∗) (Given)
Name of the axis array component whose numeric type is to be set: 'CENTRE', 'VARIANCE' or 'WIDTH'.

IAXIS = INTEGER (Given)
Number of the NDF axis whose array is to be modified.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- A comma-separated list of axis array component names may also be supplied, in which case the numeric type of each array will be set to the same value in turn.
- A value of zero may be supplied for the IAXIS argument, in which case the routine will set a new numeric type for the specified component(s) of all the NDF's axes.
- This routine may only be used to change the numeric type of an axis array via a base NDF. If an NDF section is supplied, then it will return without action. No error will result.
- The numeric type of an axis array component cannot be changed while it, or any part of it, is mapped for access (e.g. via another NDF identifier). This routine will fail, and set a STATUS value, if this is the case.
- If the numeric type of an axis array component is to be changed without its values being retained, then a call to NDF_AREST should be made beforehand. This will avoid the cost of converting all the values.

NDF_ATYPE
Obtain the numeric type of an NDF axis array

Description:

The routine returns the numeric type of an NDF axis array as an upper-case character string (e.g. '_REAL').

Invocation:

```
CALL NDF_ATYPE( INDF, COMP, IAXIS, TYPE, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the axis array component whose numeric type is required: 'CENTRE', 'VARIANCE' or 'WIDTH'.

IAXIS = INTEGER (Given)
Number of the NDF axis for which information is required.

TYPE = CHARACTER * (*) (Returned)
Numeric type of the axis array.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- A comma-separated list of axis array component names may also be supplied to this routine. In this case the result returned will be the lowest precision numeric type to which all the specified axis arrays can be converted without unnecessary loss of information.
- A value of zero may be supplied for the IAXIS argument, in which case the routine will combine the results for all the NDF's axes in the same way as described above.
- The symbolic constant NDF__SZTYP may be used for declaring the length of a character variable which is to hold the numeric type of an NDF axis array. This constant is defined in the include file NDF_PAR.

NDF_AUNMP
Unmap an NDF axis array

Description:

The routine unmaps an NDF axis array which has previously been mapped for READ, UPDATE or WRITE access.

Invocation:

```
CALL NDF_AUNMP( INDF, COMP, IAXIS, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the axis array component to be unmapped: 'CENTRE', 'VARIANCE', 'WIDTH' or '*'. The last value acts as a wild card, causing all mapped axis components to be unmapped.

IAXIS = INTEGER (Given)
Number of the NDF axis whose array is to be unmapped.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- This routine attempts to execute even if STATUS is set on entry, although no further error report will be made if it subsequently fails under these circumstances.
- A comma-separated list of axis component names may also be given, in which case each component will be unmapped in turn.
- A value of zero may be supplied for the IAXIS argument, in which case the routine will unmap the specified component(s) for all the NDF's axes.
- An error will be reported if a component has not previously been mapped for access, except in cases where a wild card unmapping operation is specified (either with a component name of '*' or an axis number of zero).

NDF_BAD
Determine if an NDF array component may contain bad pixels

Description:

The routine returns a logical value indicating whether an array component of an NDF may contain bad pixels for which checks must be made when the array's values are processed. Only if the returned value is .FALSE. can such checks be omitted. If the CHECK argument to this routine is set .TRUE., then it will also perform an explicit check (if necessary) to see whether bad pixels are actually present.

Invocation:

```
CALL NDF_BAD( INDF, COMP, CHECK, BAD, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the NDF array component: 'DATA', 'QUALITY' or 'VARIANCE'.

CHECK = LOGICAL (Given)
Whether to perform an explicit check to see whether bad pixels are actually present.

BAD = LOGICAL (Returned)
Whether it is necessary to check for bad pixels when processing the array's values.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- A comma-separated list of component names may also be supplied, in which case the routine returns the logical "OR" of the results for each component.
- If CHECK is set .FALSE., then the returned value of BAD will indicate whether bad pixels might be present and should therefore be checked for during subsequent processing. However, even if BAD is returned .TRUE. in such circumstances, it is still possible that there may not actually be any bad pixels present (for instance, in an NDF section, the accessible region of an array component might happen to avoid all the bad pixels).
- If CHECK is set .TRUE., then an explicit check will be made, if necessary, to ensure that BAD is only returned .TRUE. if bad pixels are actually present.
- If a component is mapped for access through the identifier supplied, then the value of BAD will refer to the actual mapped values. It may differ from its original (unmapped) value if conversion errors occurred during the mapping process, if an initialisation option of '/ZERO' was specified for a component whose value was initially undefined, or if the mapped values have subsequently been modified.
- A BAD=.TRUE. result will be returned for any components which are in an undefined state, except in the case of the QUALITY component for which a .FALSE. result is always returned under these circumstances.

NDF_BASE
Obtain an identifier for a base NDF

Description:

The routine returns an identifier for the base NDF with which an NDF section is associated.

Invocation:

```
CALL NDF_BASE( INDF1, INDF2, STATUS )
```

Arguments:

INDF1 = INTEGER (Given)
Identifier for an existing NDF section (the routine will also work if this is already a base NDF).

INDF2 = INTEGER (Returned)
Identifier for the base NDF with which the section is associated.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If this routine is called with STATUS set, then a value of NDF__NOID will be returned for the INDF2 argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The NDF__NOID constant is defined in the include file NDF_PAR.

NDF_BB
Obtain the bad-bits mask value for the quality component of an NDF

Description:

The routine returns an unsigned byte value representing the bad-bits mask associated with the quality component of an NDF.

Invocation:

```
CALL NDF_BB( INDF, BADBIT, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

BADBIT = BYTE (Returned)
The unsigned byte bad-bits mask.

STATUS = INTEGER (Given and Returned)
The global status.

NDF_BEGIN
Begin a new NDF context

Description:

The routine begins a new NDF context. A subsequent call to NDF_END may then be used to annul all the NDF identifiers (and placeholders) issued since the call to NDF_BEGIN was made.

Invocation:

```
CALL NDF_BEGIN
```

Notes:

Matching pairs of calls to NDF_BEGIN and NDF_END may be nested.

NDF_BLOCK
Obtain an NDF section containing a block of adjacent pixels

Description:

The routine returns an identifier for an NDF section describing a "block" of adjacent pixels selected from an initial NDF. The routine divides the original NDF logically into a series of such blocks, each of which does not exceed a specified maximum number of pixels in each dimension. The routine's IBLOCK argument allows one of these blocks to be selected; an NDF section for it is then returned.

Invocation:

```
CALL NDF_BLOCK( INDF1, NDIM, MXDIM, IBLOCK, INDF2, STATUS )
```

Arguments:

INDF1 = INTEGER (Given)
Identifier for the initial NDF.

NDIM = INTEGER (Given)
Number of maximum dimension sizes.

MXDIM(NDIM) = INTEGER (Given)
Array specifying the maximum size of a block in pixels along each dimension.

IBLOCK = INTEGER (Given)
Number of the block required (the first block is numbered 1).

INDF2 = INTEGER (Returned)
Identifier for an NDF section describing the block.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- This routine is intended to allow NDFs to be processed in smaller pieces by selecting successive blocks, each of which may then be processed individually. Note that in general not all the blocks selected from an NDF will have the same shape or size, although none will exceed the specified maximum number of pixels in each dimension.
- Corresponding blocks selected from different NDFs (or NDF sections) with identical shapes will themselves have identical shapes and will contain the same number of pixels.
- All NDF sections obtained via this routine have the same number of dimensions as the input NDF. If the number of maximum dimension sizes supplied (NDIM) is less than this number, then a value of 1 will be used for the extra dimension sizes. If the value of NDIM is larger than this number, then the excess dimension sizes will be ignored.
- If the number of the requested block (IBLOCK) exceeds the number of blocks available in the NDF, then a value of NDF__NOID will be returned for the INDF2 argument (but no error will result). This condition may be used to terminate a loop when all available blocks have been processed. The NDF_NBLOC routine may also be used to determine the number of blocks available.
- If this routine is called with STATUS set, then a value of NDF__NOID will be returned for the INDF2 argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason.
- The NDF__NOID constant is defined in the include file NDF_PAR.

NDF_BOUND
Enquire the pixel-index bounds of an NDF

Description:

The routine returns the lower and upper pixel-index bounds of each dimension of an NDF, together with the total number of dimensions.

Invocation:

```
CALL NDF_BOUND( INDF, NDIMX, LBND, UBND, NDIM, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

NDIMX = INTEGER (Given)
Maximum number of pixel-index bounds to return (i.e. the declared size of the LBND and UBND arguments).

LBND(NDIMX) = INTEGER (Returned)
Lower pixel-index bounds for each dimension.

UBND(NDIMX) = INTEGER (Returned)
Upper pixel-index bounds for each dimension.

NDIM = INTEGER (Returned)
Total number of NDF dimensions.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If the NDF has fewer than NDIMX dimensions, then any remaining elements of the LBND and UBND arguments will be filled with `1's`.
- If the NDF has more than NDIMX dimensions, then the NDIM argument will return the actual number of dimensions. In this case only the first NDIMX sets of bounds will be returned, and an error will result if the size of any of the remaining dimensions exceeds 1.
- If this routine is called with STATUS set, then a value of 1 will be returned for all elements of the LBND and UBND arrays and for the NDIM argument, although no further processing will occur. The same values will also be returned if the routine should fail for any reason.
- The symbolic constant NDF__MXDIM may be used to declare the size of the LBND and UBND arguments so that they will be able to hold the maximum number of NDF bounds that this routine can return. This constant is defined in the include file NDF_PAR.

NDF_CANCL
Cancel the association of an NDF with an ADAM parameter

Description:

This routine cancels the association of an NDF with an ADAM parameter. A subsequent attempt to get a value for the parameter will result in a new value being obtained by the underlying parameter system.

By supplying a blank parameter name, all currently active NDF parameters can be cancelled in a single call. However, it is possible to exclude selected parameters from this automatic cancellation if necessary. To do this, the parameter to be excluded should be marked by making a prior call to this routine with an asterisk appended to the end of the parameter name. Any subsequent call to this routine with a blank parameter name will skip such marked parameters. To mark all currently active NDF parameters in this way, supply the PARAM argument holding just an asterisk.

Invocation:

```
CALL NDF_CANCL( PARAM, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)
Name of the ADAM parameter to be cancelled or marked.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- When cancelling a parameter, the behaviour of this routine is identical to PAR_CANCL.
- Any remaining NDF identifiers for the associated NDF are unaffected by this routine. It's only affect is to cause NDF_ASSOC or NDF_EXIST to prompt for a new NDF when called subsequently.
- This routine attempts to execute even if STATUS is set on entry, although no further error report will be made if it subsequently fails under these circumstances.

NDF_CGET
Obtain the value of an NDF character component

Description:

The routine obtains the value of the specified character component of an NDF (i.e. the value of the LABEL, TITLE or UNITS component).

Invocation:

```
CALL NDF_CGET( INDF, COMP, VALUE, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the character component whose value is required: 'LABEL', 'TITLE' or 'UNITS'.

VALUE = CHARACTER * (*) (Given and Returned)
The component's value.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If the requested component is in an undefined state, then the VALUE argument will be returned unchanged. A suitable default should therefore be established before calling this routine.
- If the length of the VALUE argument is too short to accommodate the returned result without losing significant (non-blank) trailing characters, then this will be indicated by an appended ellipsis, i.e. '...'. No error will result.

NDF_CHUNK
Obtain an NDF section containing a chunk of contiguous pixels

Description:

The routine returns an identifier for an NDF section describing a "chunk" of contiguous pixels selected from an initial NDF. The routine divides the initial NDF logically into a series of such chunks, each of which follows immediately on from the previous chunk, and each of which contains no more than a specified maximum number (MXPIX) of contiguous pixels. The routine's ICHUNK argument allows one of these chunks to be selected; an NDF section for it is then returned.

Invocation:

```
CALL NDF_CHUNK( INDF1, MXPIX, ICHUNK, INDF2, STATUS )
```

Arguments:

INDF1 = INTEGER (Given)
Identifier for the initial NDF.

MXPIX = INTEGER (Given)
Maximum number of contiguous pixels required in each chunk.

ICHUNK = INTEGER (Given)
Number of the chunk required (the first chunk is numbered 1).

INDF2 = INTEGER (Returned)
Identifier for an NDF section describing the chunk.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- This routine is intended to allow large NDFs to be processed in smaller pieces by selecting successive chunks, each of which may then be processed individually. Note that in general not all the chunks selected from an NDF will have the same size, although none will contain more than the specified maximum number of pixels.
- Corresponding chunks selected from different NDFs (or NDF sections) with identical shapes will themselves have identical shapes and will contain the same number of pixels.
- All NDF sections obtained via this routine have the same number of dimensions as the input NDF.
- If the number of the requested chunk (ICHUNK) exceeds the number of chunks available in the NDF, then a value of NDF__NOID will be returned for the INDF2 argument (but no error will result). This condition may be used to terminate a loop when all available chunks have been processed. The NDF_NCHNK routine may also be used to determine the number of chunks available.
- If this routine is called with STATUS set, then a value of NDF__NOID will be returned for the INDF2 argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason.
- The NDF__NOID constant is defined in the include file NDF_PAR.

NDF_CINP
Obtain an NDF character component value via the ADAM parameter system

Description:

The routine obtains a new value for a character component of an NDF via the ADAM parameter system and uses it to replace any pre-existing value of that component in the NDF.

Invocation:

```
CALL NDF_CINP( PARAM, INDF, COMP, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)
Name of the ADAM parameter.

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the character component for which a value is to be obtained: 'LABEL', 'TITLE' or 'UNITS'.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- A "null" parameter value is interpreted as indicating that no new value should be set for the character component. In this event, the routine will return without action (and without setting a STATUS value). A suitable default value for the character component should therefore be established before this routine is called.

NDF_CLEN
Determine the length of an NDF character component

Description:

The routine returns the length of the specified character component of an NDF (i.e. the number of characters in the LABEL, TITLE or UNITS component).

Invocation:

```
CALL NDF_CLEN( INDF, COMP, LENGTH, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the character component whose length is required: 'LABEL', 'TITLE' or 'UNITS'.

LENGTH = INTEGER (Returned)
Length of the component in characters.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- The length of an NDF character component is determined by the length of the VALUE string assigned to it by a previous call to NDF_CPUT (note that this could include trailing blanks).
- If the specified component is in an undefined state, then a length of zero will be returned.

NDF_CLONE
Clone an NDF identifier

Description:

The routine produces a "cloned" copy of an NDF identifier (i.e. it produces a new identifier describing an NDF with identical attributes to the original).

Invocation:

```
CALL NDF_CLONE( INDF1, INDF2, STATUS )
```

Arguments:

INDF1 = INTEGER (Given)
NDF identifier to be cloned.

INDF2 = INTEGER (Returned)
Cloned identifier.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If this routine is called with STATUS set, then a value of NDF__NOID will be returned for the INDF2 argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The NDF__NOID constant is defined in the include file NDF_PAR.

NDF_CMPLX
Determine whether an NDF array component holds complex values

Description:

The routine returns a logical value indicating whether the specified array component of an NDF holds complex values.

Invocation:

```
CALL NDF_CMPLX( INDF, COMP, CMPLX, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the NDF array component: 'DATA', 'QUALITY' or 'VARIANCE'.

CMPLX = LOGICAL (Returned)
Whether the component holds complex values.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- A comma-separated list of components may also be specified, in which case the logical "OR" of the results for each component will be returned.
- The value returned for the QUALITY component is always .FALSE..

NDF_CMSG
Assign the value of an NDF character component to a message token

Description:

The routine assigns the value of the specified character component of an NDF to a message token, for use in constructing messages using the MSG_ or ERR_ routines (see SUN/104).

Invocation:

```
CALL NDF_CMSG( TOKEN, INDF, COMP, STATUS )
```

Arguments:

TOKEN = CHARACTER * (*) (Given)
Name of the message token.

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the character component whose value is to be used: 'LABEL', 'TITLE' or 'UNITS'.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If the specified NDF component does not have a defined value, then the string '<undefined>' is assigned to the token instead.

NDF_COPY
Copy an NDF to a new location

Description:

The routine copies an NDF to a new location and returns an identifier for the resulting new base NDF.

Invocation:

```
CALL NDF_COPY( INDF1, PLACE, INDF2, STATUS )
```

Arguments:

INDF1 = INTEGER (Given)
Identifier for the NDF (or NDF section) to be copied.

PLACE = INTEGER (Given and Returned)
An NDF placeholder (e.g. generated by the NDF_PLACE routine) which indicates the position in the data system where the new NDF will reside. The placeholder is annulled by this routine, and a value of NDF__NOPL will be returned (as defined in the include file NDF_PAR).

INDF2 = INTEGER (Returned)
Identifier for the new NDF.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If this routine is called with STATUS set, then a value of NDF__NOID will be returned for the INDF2 argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. In either event the placeholder will still be annulled. The NDF__NOID constant is defined in the include file NDF_PAR.

NDF_CPUT
Assign a value to an NDF character component

Description:

The routine assigns a value to the specified character component of an NDF (i.e. to the LABEL, TITLE or UNITS component). Any previous value is over-written.

Invocation:

```
CALL NDF_CPUT( VALUE, INDF, COMP, STATUS )
```

Arguments:

VALUE = CHARACTER * (*) (Given)
The value to be assigned.

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the character component whose value is to be assigned: 'LABEL', 'TITLE' or 'UNITS'.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- The entire VALUE string (including trailing blanks if present) is assigned to the specified component, whose length is adjusted to accommodate it.

NDF_CREAT
Create a new simple NDF via the ADAM parameter system

Description:

The routine creates a new simple NDF via the ADAM parameter system, associates it with a parameter, and returns an NDF identifier for it.

Invocation:

```
CALL NDF_CREAT( PARAM, FTYPE, NDIM, LBND, UBND, INDF, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)
Name of the ADAM parameter.

FTYPE = CHARACTER * (*) (Given)
Full data type of the NDF's DATA component (e.g. '_DOUBLE' or 'COMPLEX_REAL').

NDIM = INTEGER (Given)
Number of NDF dimensions.

LBND(NDIM) = INTEGER (Given)
Lower pixel-index bounds of the NDF.

UBND(NDIM) = INTEGER (Given)
Upper pixel-index bounds of the NDF.

INDF = INTEGER (Returned)
NDF identifier.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- This routine creates a "simple" NDF, i.e. one whose array components will be stored in "simple" form by default (see SGP/38).
- The full data type of the DATA component is specified via the FTYPE argument and the data type of the VARIANCE component defaults to the same value. These data types may be set individually with the NDF_STYPE routine if required.
- If this routine is called with STATUS set, then a value of NDF__NOID will be returned for the INDF argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The NDF__NOID constant is defined in the include file NDF_PAR.

NDF_CREP
Create a new primitive NDF via the ADAM parameter system

Description:

The routine creates a new primitive NDF via the ADAM parameter system, associates it with a parameter, and returns an NDF identifier for it.

Invocation:

```
CALL NDF_CREP( PARAM, FTYPE, NDIM, UBND, INDF, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)
Name of the ADAM parameter.

FTYPE = CHARACTER * (*) (Given)
Type of the NDF's DATA component (e.g. '_REAL'). Note that complex types are not permitted when creating a primitive NDF.

NDIM = INTEGER (Given)
Number of NDF dimensions.

UBND(NDIM) = INTEGER (Given)
Upper pixel-index bounds of the NDF (the lower bound of each dimension is taken to be 1).

INDF = INTEGER (Returned)
NDF identifier.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- This routine creates a "primitive" NDF, i.e. one whose array components will be stored in "primitive" form by default (see SGP/38).
- The data type of the DATA component is specified via the FTYPE argument and the data type of the VARIANCE component defaults to the same value. These data types may be set individually with the NDF_STYPE routine if required.
- If this routine is called with STATUS set, then a value of NDF__NOID will be returned for the INDF argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The NDF__NOID constant is defined in the include file NDF_PAR.

NDF_CREPL
Create a new NDF placeholder via the ADAM parameter system

Description:

The routine creates a new NDF placeholder via the ADAM parameter system, associates it with a parameter, and returns an identifier for it. A placeholder is used to identify a position in the underlying data system (HDS) and may be passed to other routines (e.g. NDF_NEW) to indicate where a newly created NDF should be positioned.

Invocation:

```
CALL NDF_CREPL( PARAM, PLACE, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)
Name of the ADAM parameter.

PLACE = INTEGER (Returned)
NDF placeholder identifying the nominated position in the data system.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- Placeholders are intended only for local use within an application and only a limited number of them are available simultaneously. They are always annulled as soon as they are passed to another routine to create a new NDF, where they are effectively exchanged for an NDF identifier.
- If this routine is called with STATUS set, then a value of NDF__NOPL will be returned for the PLACE argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The NDF__NOPL constant is defined in the include file NDF_PAR.

NDF_DELET
Delete an NDF

Description:

The routine deletes the specified NDF. If this is a base NDF, then the associated data object is erased and all NDF identifiers which refer to it (or to sections derived from it) become invalid. If any NDF components are mapped for access, then they are first unmapped. If an NDF section is specified, then this routine is equivalent to calling NDF_ANNUL, and no other identifiers are affected.

Invocation:

```
CALL NDF_DELET( INDF, STATUS )
```

Arguments:

INDF = INTEGER (Given and Returned)

Identifier for the NDF to be deleted. A value of NDF__NOID is returned (as defined in the include file NDF_PAR).

STATUS = INTEGER (Given and Returned)

The global status.

Notes:

- This routine attempts to execute even if STATUS is set on entry, although no further error report will be made if it subsequently fails under these circumstances.

NDF_DIM
Enquire the dimension sizes of an NDF

Description:

The routine returns the size in pixels of each dimension of an NDF, together with the total number of dimensions (the size of a dimension is the difference between that dimension's upper and lower pixel-index bounds + 1).

Invocation:

```
CALL NDF_DIM( INDF, NDIMX, DIM, NDIM, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

NDIMX = INTEGER (Given)
Maximum number of dimension sizes to return (i.e. the declared size of the DIM argument).

DIM(NDIMX) = INTEGER (Returned)
Size of each dimension in pixels.

NDIM = INTEGER (Returned)
Total number of NDF dimensions.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If the NDF has fewer than NDIMX dimensions, then any remaining elements of the DIM argument will be filled with 1's.
- If the NDF has more than NDIMX dimensions, then the NDIM argument will return the actual number of dimensions. In this case only the first NDIMX dimension sizes will be returned, and an error will result if the size of any of the excluded dimensions exceeds 1.
- If this routine is called with STATUS set, then a value of 1 will be returned for all elements of the DIM array and for the NDIM argument, although no further processing will occur. The same values will also be returned if the routine should fail for any reason.
- The symbolic constant NDF__MXDIM may be used to declare the size of the DIM argument so that it will be able to hold the maximum number of NDF dimension sizes that this routine can return. This constant is defined in the include file NDF_PAR.

NDF_END
End the current NDF context

Description:

The routine ends the current NDF context, causing all NDF identifiers and placeholders created within that context (i.e. since a matching call to NDF_BEGIN) to be annulled. Any mapped values associated with these identifiers are unmapped, and any temporary NDFs which no longer have identifiers associated with them are deleted.

Invocation:

```
CALL NDF_END( STATUS )
```

Arguments:

STATUS = INTEGER (Given and Returned)

The global status.

Notes:

- Matching pairs of calls to NDF_BEGIN and NDF_END may be nested. An error will be reported if NDF_END is called without a corresponding call to NDF_BEGIN.
- This routine attempts to execute even if STATUS is set on entry, although no further error report will be made if it subsequently fails under these circumstances.

NDF_EXIST
See if an existing NDF is associated with an ADAM parameter

Description:

The routine determines if an existing (and accessible) NDF is associated with an ADAM parameter. If it is, then an identifier is returned for it. If not, then the routine returns with an identifier value of NDF__NOID; this then allows the NDF structure to be created (e.g. using NDF_CREAT) if required.

Invocation:

```
CALL NDF_EXIST( PARAM, MODE, INDF, STATUS )
```

Arguments:

PARAM = CHARACTER * (*) (Given)
Name of the ADAM parameter.

MODE = CHARACTER * (*) (Given)
Type of NDF access required: 'READ', 'UPDATE' or 'WRITE'.

INDF = INTEGER (Returned)
NDF identifier.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If 'WRITE' access is specified, then all the NDF's components will be reset to an undefined state ready to receive new values. If 'UPDATE' access is specified, the NDF's components will retain their values, which may then be modified.
- The behaviour of this routine is the same as NDF_ASSOC, except that in the event of the NDF structure not existing (or being inaccessible), control is returned to the application with an identifier value of NDF__NOID, rather than re-prompting the user.
- Note that unlike the DAT_EXIST routine, on which it is modelled, this routine does not set a STATUS value if the data structure does not exist.
- If this routine is called with STATUS set, then a value of NDF__NOID will be returned for the INDF argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The NDF__NOID constant is defined in the include file NDF_PAR.

NDF_FIND
Find an NDF and import it into the NDF_ system

Description:

The routine finds an NDF within an HDS structure or container file, imports it into the NDF_ system and issues an identifier for it. The imported NDF may then be manipulated by the NDF_ routines.

Invocation:

```
CALL NDF_FIND( LOC, NAME, INDF, STATUS )
```

Arguments:

LOC = CHARACTER * (*) (Given)
Locator to the enclosing HDS structure.

NAME = CHARACTER * (*) (Given)
Name of the HDS structure component to be imported.

INDF = INTEGER (Returned)
NDF identifier.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- The value given for the NAME argument may be an HDS path name, consisting of several fields separated by '.', so that an NDF can be found in a sub-component (or a sub-sub-component...) of the structure identified by the locator LOC. Array subscripts may also be used in this component name. Thus a string such as 'MYSTRUC.ZONE(2).IMAGE' could be used as a valid NAME value.
- An NDF can be accessed within an explicitly named container file by supplying the symbolic value DAT__ROOT for the LOC argument and specifying the container file within the value supplied for the NAME argument. Only READ access is available to an NDF accessed in this way (for other modes of access, see the NDF_OPEN routine).
- If a blank value is given for the NAME argument, then the NDF to be imported will be the object identified directly by the locator LOC.
- The locator supplied as input to this routine may later be annulled without affecting the behaviour of the NDF_ system.
- If this routine is called with STATUS set, then a value of NDF__NOID will be returned for the INDF argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason.
- The NDF__NOID constant is defined in the include file NDF_PAR. The DAT__ROOT constant is defined in the include file DAT_PAR (see SUN/92).

NDF_FORM
Obtain the storage form of an NDF array component

Description:

The routine returns the storage form of an NDF array component as an upper case character string (e.g. 'SIMPLE').

Invocation:

```
CALL NDF_FORM( INDF, COMP, FORM, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the NDF array component whose storage form is required: 'DATA', 'QUALITY' or 'VARIANCE'.

FORM = CHARACTER * (*) (Returned)
Storage form of the component.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- The symbolic constant NDF__SZFRM may be used for declaring the length of a character variable to hold the storage form of an NDF array component. This constant is defined in the include file NDF_PAR.
- At present, the NDF_ routines only support "primitive", "simple", "delta" and "scaled" arrays, so only the values 'PRIMITIVE', 'SIMPLE', 'DELTA' and 'SCALED' can be returned.

NDF_FTYPE
Obtain the full type of an NDF array component

Description:

The routine returns the full data type of one of the array components of an NDF as an upper-case character string (e.g. '_REAL' or 'COMPLEX_BYTE').

Invocation:

```
CALL NDF_FTYPE( INDF, COMP, FTYPE, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the NDF array component whose type is required: 'DATA', 'QUALITY' or 'VARIANCE'.

FTYPE = CHARACTER * (*) (Returned)
Full data type of the component.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- A comma-separated list of component names may also be supplied to this routine. In this case the result returned will be the lowest precision full data type to which the values held in all the specified components can be converted without unnecessary loss of information.
- The numeric type of a scaled array is determined by the numeric type of the scale and zero terms, not by the numeric type of the underlying array elements.
- The value returned for the QUALITY component is always '_UBYTE'.
- The symbolic constant NDF__SZFTP may be used for declaring the length of a character variable to hold the full data type of an NDF array component. This constant is defined in the include file NDF_PAR.

NDF_GTDLT
Get compression details for a DELTA compressed NDF array component

Description:

The routine returns the details of the compression used by an NDF array component stored in DELTA form. If the array is not stored in DELTA form, then null values are returned as listed below, but no error is reported.

A DELTA array is compressed by storing only the differences between adjacent array values along a nominated compression axis, rather than the full array values. The differences are stored using a smaller data type than the original absolute values. The compression is lossless because any differences that will not fit into the smaller data type are stored explicitly in an extra array with a larger data type. Additional compression is achieved by replacing runs of equal values by a single value and a repeat count.

Invocation:

```
CALL NDF_GTDLT( INDF, COMP, ZAXIS, ZTYPE, ZRATIO, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the NDF array component; 'DATA', 'QUALITY' or 'VARIANCE'.

ZAXIS = INTEGER (Returned)
The index of the pixel axis along which compression occurred. The first axis has index 1. Zero is returned if the array is not stored in DELTA form.

ZTYPE = CHARACTER * (*) (Returned)
The data type in which the differences between adjacent array values are stored. This will be one of '_BYTE', '_WORD' or '_INTEGER'. The data type of the array itself is returned if the supplid array is not stored in DELTA form.

ZRATIO = REAL (Returned)
The compression factor - the ratio of the uncompressed array size to the compressed array size. This is approximate as it does not include the effects of the metadata needed to describe the extra components of a DELTA array (i.e. the space needed to hold the component names, types, dimensions, etc). A value of 1.0 is returned if the supplid array is not stored in DELTA form.

STATUS = INTEGER (Given and Returned)
The global status.

NDF_GTSZx
Get the scale and zero values for an NDF array component

Description:

The routine returns the scale and zero values associated with an NDF array component. If the array is stored in simple or primitive form, then values of 1.0 and 0.0 are returned.

Invocation:

```
CALL NDF_GTSZx( INDF, COMP, SCALE, ZERO, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the NDF array component; 'DATA' or 'VARIANCE'.

SCALE = ? (Returned)
The new value for the scaling factor.

ZERO = ? (Returned)
The new value for the zero offset.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- There is a routine for each of the standard Fortran numerical data types: integer, real and double precision. Replace the (lower case) "x" in the routine name by I, R or D as appropriate.

NDF_GTUNE
Obtain the value of an NDF_ system tuning parameter

Description:

The routine returns the current value of an NDF_ system internal tuning parameter.

Invocation:

```
CALL NDF_GTUNE( TPAR, VALUE, STATUS )
```

Arguments:

TPAR = CHARACTER * (*) (Given)

Name of the tuning parameter whose value is required (case insensitive). This name may be abbreviated, to no less than 3 characters.

VALUE = INTEGER (Returned)

Value of the parameter.

STATUS = INTEGER (Given and Returned)

The global status.

Notes:

See the NDF_TUNE routine for a list of the tuning parameters currently available.

NDF_GTWCS
Obtain world coordinate system information from an NDF

Description:

The routine obtains information about the world coordinate systems associated with an NDF and returns an AST pointer to a FrameSet which contains this information. The information may then be accessed using routines from the AST library (SUN/210).

Invocation:

```
CALL NDF_GTWCS( INDF, IWCS, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

IWCS = INTEGER (Returned)
An AST pointer to a FrameSet which contains information about the world coordinate systems associated with the NDF.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- It is the caller's responsibility to annul the AST pointer issued by this routine (e.g. by calling AST_ANNUL) when it is no longer required. The NDF_ system will not perform this task itself.
- If this routine is called with STATUS set, then a value of AST__NULL will be returned for the IWCS argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The AST__NULL constant is defined in the include file AST_PAR.

NDF_HAPPN
Declare a new application name for NDF history recording

Description:

The routine declares a new application name to be used as the default for subsequent recording of NDF history information. The name supplied will subsequently be used when creating new history records whenever a blank application name is passed to a routine which writes new history information. It will also be used as the application name when recording default history information.

If this routine is not called, then a system-supplied default name will be used in its place.

Invocation:

```
CALL NDF_HAPPN( APPN, STATUS )
```

Arguments:

APPN = CHARACTER * (*) (Given)
Name of the new application. If a blank value is supplied, then the name will revert to the system-supplied default.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- This routine should normally only be called to set up an application name in cases where better information is available than is provided by the default. For example, writers of environment-level software may be able to include a software version number in the name so that individual applications need not duplicate this in their own calls to NDF history routines.
- The maximum number of application name characters which can be stored by this routine is given by the constant NDF__SZAPP. The name supplied will be truncated without error if more then this number of characters are supplied. The NDF__SZAPP constant is defined in the include file NDF_PAR.

NDF_HCOPY
Copy history information from one NDF to another

Description:

The routine copies history information from one NDF to another, replacing any that already exists in the destination NDF.

Invocation:

```
CALL NDF_HCOPY( INDF1, INDF2, STATUS )
```

Arguments:

INDF1 = INTEGER (Given)
Identifier for the NDF (or NDF section) containing the history information to be copied.

INDF2 = INTEGER (Given)
Identifier for the NDF to receive the copied history information.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- This routine returns without action leaving the destination NDF unchanged if no History component exists in the input NDF.
- If the input NDF contains a History component, then a History component is added to the destination NDF automatically, if one does not already exist.

NDF_HCRE
Ensure that a history component exists for an NDF

Description:

The routine ensures that an NDF has a history component, creating a new one if necessary. No action is taken if a history component already exists.

Invocation:

```
CALL NDF_HCRE( INDF, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

A history component may be removed from an NDF by calling NDF_RESET with a component name of 'History'.

NDF_HDEF
Write default history information to an NDF

Description:

The routine writes default information about the current application to the history component of an NDF, creating a new history record if necessary.

Invocation:

```
CALL NDF_HDEF( INDF, APPN, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

APPN = CHARACTER * (*) (Given)
Name of the current application. This will only be used if a new history record is created by this routine, otherwise it is ignored. If a blank value is given, then a suitable default will be used instead.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- The values stored in the history component for any program parameters that have not yet been accessed by the program at the time this routine is called may not be correct. The value left by the previous invocation of the program will be used, if it is stored in the program's parameter file. If there is no value for the parameter in the parameter file, a value of "<not yet accessed>" will be stored for the parameter in the history component. For this reason, this routine should usually be called once all program parameters have been accessed.
- Default history information will normally be provided automatically by the NDF_ system when an NDF is released, so a call to this routine is not usually required. It is provided so that premature writing of default history information can be forced in cases where additional text will then be appended to it (using NDF_HPUT).
- It is expected that the APPN argument will usually be left blank. A non-blank value should normally only be given if a more complete identification of the current application can be given than is supplied by default.
- This routine will return without action if (a) there is no history component present in the NDF, (b) the NDF's history update mode is currently 'DISABLED', (c) default history information has already been written to the NDF, or (d) a previous call has been made to NDF_HPUT specifying that default history information is to be replaced.

NDF_HECHO
Write out lines of history text

Description:

The routine writes a series of lines of text to the standard output channel, indented by three spaces. It is provided as a default service routine which may be passed as an argument to NDF_HOUT in order to display NDF history information.

The specification of this routine may be used as a template when writing alternative service routines for use by NDF_HOUT.

Invocation:

```
CALL NDF_HECHO( NLINES, TEXT, STATUS )
```

Arguments:

NLINES = INTEGER (Given)
Number of lines of text to be written.

TEXT(NLINES) = CHARACTER * (*) (Given)
Array of text lines.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

Before passing this (or a similar) routine as an argument to NDF_HOUT the calling routine should declare it in a Fortran EXTERNAL statement.

NDF_HEND
End NDF history recording for the current application

Description:

The routine closes down history recording for the current application by writing default history information (when appropriate) to any NDFs which are still active, and by flagging that a new history record should be created to hold any subsequent history information (ready for the next application). If an application name has been declared via a call to NDF_HAPPN, then that name is cleared.

Invocation:

```
CALL NDF_HEND( STATUS )
```

Arguments:

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- This routine attempts to execute even if STATUS is set on entry, although no further error report will be made if it subsequently fails under these circumstances.
- This routine should not normally be used by writers of NDF applications. It is provided primarily to allow those writing environment-level software to identify the end of an application to the NDF_ system in circumstances where more than one application may be invoked from within a single executing program. Even then, it will not normally be needed unless NDF data structures are intended to remain open throughout the invocation of several applications and a separate history record is required from each application.

NDF_HFIND
Find an NDF history record by date and time

Description:

The routine searches the history component of an NDF to identify the first history record which was written after a specified date and time. The record number is returned. A value of zero is returned if no suitable record exists.

Invocation:

```
CALL NDF_HFIND( INDF, YMDHM, SEC, EQ, IREC, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

YMDHM(5) = INTEGER (Given)
The year, month, day, hour and minute fields of the required date and time, in that order, stored as integers (the month field starts at 1 for January).

SEC = REAL (Given)
The seconds field of the required date and time.

EQ = LOGICAL (Given)
If a .TRUE. value is given for this argument, then a history record whose date and time exactly matches that specified may be returned. Otherwise, the record must have been written strictly later than specified.

IREC = INTEGER (Returned)
The record number of the required history record, or zero if no suitable record exists.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

The last history record written before a specified date and time may be found by subtracting 1 from the record number returned by this routine (or using the final record if this routine returns zero).

NDF_HGMOD
Get the history update mode for an NDF

Description:

The routine returns the current history component update mode of an NDF. See NDF_HSMOD.

Invocation:

```
CALL NDF_HGMOD( INDF, HMODE, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

HMODE = CHARACTER * (*) (Returned)
The history update mode: 'DISABLED', 'QUIET', 'NORMAL' or 'VERBOSE'.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- An error is reported if the NDF has no HISTORY component.

NDF_HINFO
Obtain information about an NDF's history component

Description:

The routine returns character information about an NDF's history component or about one of the history records it contains.

Invocation:

```
CALL NDF_HINFO( INDF, ITEM, IREC, VALUE, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

ITEM = CHARACTER * (*) (Given)
Name of the information item required: 'APPLICATION', 'CREATED', 'DATE', 'DEFAULT', 'HOST', 'MODE', 'NLINES', 'NRECORDS', 'REFERENCE', 'USER', 'WIDTH' or 'WRITTEN' (see the "General Items" and "Specific Items" sections for details). This value may be abbreviated, to no less than three characters.

IREC = INTEGER (Given)
History record number for which information is required. This argument is ignored if information is requested about the history component as a whole. See the "Specific Items" section for details of which items require this argument.

VALUE = CHARACTER * (*) (Returned)
The history information requested (see the "Returned String Lengths" section for details of the length of character variable required to receive this value).

STATUS = INTEGER (Given and Returned)
The global status.

General Items :

The following ITEM values request general information about the history component and do not use the IREC argument:

- 'CREATED': return a string giving the date and time of creation of the history component as a whole in the format 'YYYY-MMM-DD HH:MM:SS.SSS' (e.g. '1993-JUN-16 11:30:58.001').
- 'DEFAULT': return a logical value indicating whether default history information has yet to be written for the current application. A value of 'F' is returned if it has already been written or has been suppressed by a previous call to NDF_HPUT, otherwise the value 'T' is returned.
- 'MODE': return the current update mode of the history component (one of the strings 'DISABLED', 'QUIET', 'NORMAL' or 'VERBOSE').
- 'NRECORDS': return the number of history records present (an integer formatted as a character string). Note that for convenience this value may also be obtained directly as an integer via the routine NDF_HNREC.
- 'WRITTEN': return a logical value indicating whether the current application has written a new history record to the NDF's history component. A value of 'T' is returned if a new record has been written, otherwise 'F' is returned.

Specific Items :

The following ITEM values request information about specific history records and should be accompanied by a valid value for the IREC argument specifying the record for which information is required:

- 'APPLICATION': return the name of the application which created the history record.
- 'DATE': return a string giving the date and time of creation of the specified history record in the format 'YYYY-MMM-DD HH:MM:SS.SSS' (e.g. '1993-JUN-16 11:36:09.021').
- 'HOST': return the name of the machine on which the application which wrote the history record was running (if this has not been recorded, then a blank value is returned).
- 'NLINES': return the number of lines of text contained in the history record (an integer formatted as a character string).
- 'REFERENCE': return a name identifying the NDF dataset in which the history component resided at the time the record was written (if this has not been recorded, then a blank value is returned). This value is primarily of use in identifying the ancestors of a given dataset when history information has been repeatedly propagated through a sequence of processing steps.
- 'USER': return the user name for the process which wrote the history record (if this has not been recorded, then a blank value is returned).
- 'WIDTH': return the width in characters of the text contained in the history record (an integer formatted as a character string).

Returned String Lengths :

- If ITEM is set to 'CREATED', 'DATE', 'MODE', 'NLINES', 'NRECORDS' or 'WIDTH', then an error will result if the length of the VALUE argument is too short to accommodate the returned result without losing significant (non-blank) trailing characters.
- If ITEM is set to 'APPLICATION', 'HOST', 'REFERENCE' or 'USER', then the returned value will be truncated with an ellipsis '...' if the length of the VALUE argument is too short to accommodate the returned result without losing significant (non-blank) trailing characters. No error will result.
- When declaring the length of character variables to hold the returned result, the constant NDF__SZHDT may be used for the length of returned date/time strings for the 'CREATED' and 'DATE' items, the constant NDF__SZHUM may be used for the length of returned update mode strings for the 'MODE' item, and the constant VAL__SZI may be used for the length of returned integer values formatted as character strings.
- Use of the constant NDF__SZAPP is recommended when declaring the length of a character variable to hold the returned application name for the 'APPLICATION' item. Similarly, use of the constant NDF__SZHST is recommended when requesting the 'HOST' item, NDF__SZREF when requesting the 'REFERENCE' item and NDF__SZUSR when requesting the 'USER' item. Truncation of the returned values may still occur, however, if longer strings were specified when the history record was created.
- The NDF__SZAPP, NDF__SZHDT, NDF__SZHST, NDF__SZHUM, NDF__SZREF and NDF__SZUSR constants are defined in the include file NDF_PAR. The VAL__SZI constant is defined in the include file PRM_PAR (see SUN/39).

NDF_HNREC
Determine the number of NDF history records present

Description:

The routine returns a count of the number of history records present in an NDF.

Invocation:

```
CALL NDF_HNREC( INDF, NREC, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

NREC = INTEGER (Returned)
Number of history records.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- The number of records returned may be zero if a history component exists but no history information has yet been entered.
- An error will result if there is no history component present in the NDF.

NDF_HOUT
Display text from an NDF history record

Description:

The routine displays the text associated with a specified NDF history record by invoking a service routine suppled by the caller. A standard service routine NDF_HECHO is provided, but this may be replaced if required.

Invocation:

```
CALL NDF_HOUT( INDF, IREC, ROUTIN, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

IREC = INTEGER (Given)
Number of the NDF history record whose text is to be displayed.

ROUTIN = SUBROUTINE (Given)
A service routine to which the text will be passed for display. For a specification of this routine, see the default routine NDF_HECHO. This service routine must be declared EXTERNAL in the routine which invokes NDF_HOUT.

STATUS = INTEGER (Given and Returned)
The global status.

NDF_HPURG
Delete a range of records from an NDF history component

Description:

The routine deletes a specified range of records from an NDF history component. The remaining records are re-numbered starting from 1.

Invocation:

```
CALL NDF_HPURG( INDF, IREC1, IREC2, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

IREC1 = INTEGER (Given)
Number of the first history record to be deleted.

IREC2 = INTEGER (Given)
Number of the last history record to be deleted.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- This routine is provided primarily to allow a lengthy NDF history to be truncated in order to save space. To avoid deceiving subsequent readers, it is normally advisable not to delete arbitrary sections of an NDF's history, but to delete only the earliest part (by setting IREC1 to 1).
- The IREC1 and IREC2 arguments must both identify valid history records which are actually present. Their values may be interchanged without affecting the behaviour of this routine.

NDF_HPUT
Write history information to an NDF

Description:

The routine writes textual information to the history component of an NDF, creating a new history record if necessary. A variety of formatting options are available and values may be substituted for message tokens embedded within the text. The text supplied may either augment or replace the history information normally written by default.

Invocation:

```
CALL NDF_HPUT( HMODE, APPN, REPL, NLINES, TEXT, TRANS, WRAP, RJUST, INDF,
STATUS )
```

Arguments:

HMODE = CHARACTER * (*) (Given)
The priority for the text being written: 'VERBOSE', 'NORMAL' or 'QUIET', where 'VERBOSE' signifies the lowest priority and 'QUIET' signifies the highest. The value given may be abbreviated, to no less then three characters. A blank value is accepted as a synonym for 'NORMAL'.

APPN = CHARACTER * (*) (Given)
Name of the current application. This will only be used if a new history record is created by this routine, otherwise it is ignored. If a blank value is given, then a system-supplied default will be used instead.

REPL = LOGICAL (Given)
Whether the text supplied is intended to replace the history information which is supplied by default. If a .TRUE. value is given and no default history information has yet been written to the NDF, then subsequent writing of this information will be suppressed. If a .FALSE. value is given, then default history information is not suppressed and will later be appended to the text supplied (unless suppressed by another call to NDF_HPUT).

NLINES = INTEGER (Given)
Number of lines of history text supplied.

TEXT(NLINES) = CHARACTER * (*) (Given)
The lines of history text to be written. The length of the elements of this array (as returned by the Fortran LEN function) should not exceed the value NDF__SZHMX. The recommended length of TEXT elements is given by the constant NDF__SZHIS. (Both of these constants are defined in the include file NDF_PAR.)

TRANS = LOGICAL (Given)
If a .TRUE. value is supplied, then any message tokens embedded in the supplied text will be expanded before it is written to the history record (see SUN/104 for a description of how to use message tokens). If a .FALSE. value is given, then the supplied text is taken literally; no special characters will be recognised and no message token expansion will occur.

WRAP = LOGICAL (Given)
If a .TRUE. value is given, then paragraph wrapping will be performed on the supplied text (after message token expansion if appropriate) so as to make as much text fit on to each line of a history record as possible. Blank input lines may be used to delimit paragraphs. If a .FALSE. value is given, then input lines which exceed the history record's text width will simply be broken (at a space if possible) and continued on a new line.

RJUST = LOGICAL (Given)
If a .TRUE. value is given, then lines of history text will be padded out with blanks (after message token expansion and paragraph wrapping if appropriate) so as to give a justified right margin. If a .FALSE. value is given, then the right margin will remain ragged.

INDF = INTEGER (Given)
NDF identifier.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- This routine will return without action if (a) there is no history component present in the NDF, (b) the priority specified via HMODE is lower than the NDF's current history update mode setting, or (c) the NDF's history update mode is currently set to 'DISABLED'.
- It is expected that the APPN argument will usually be left blank. A non-blank value should normally only be given for this argument if a more complete identification of the current application can be given than is supplied by default.
- If no previous history information has been written to the NDF by the current application, then this routine will create a new history record whose text width will be determined by the length of the lines of the TEXT array (as returned by the Fortran LEN function). If history information has already been written, then this routine will append to the existing history record. In this case, the text width will already have been defined, so the supplied text will be re-formatted if necessary (by line breaking or paragraph wrapping) to fit into the available width.
- Paragraph wrapping is recommended, when appropriate, as a means of saving space while retaining a neat appearance in the resulting history record. It is particularly useful when message tokens are used within normal text, since these make it hard to predict the precise length of history lines in advance. The right justification flag may be used to improve the cosmetic appearance of history text, but it has no effect on the amount of space used.
- On exit from this routine, all message tokens in the current message context are left in an undefined state.

NDF_HSDAT
Set the history date for an NDF

Description:

The routine sets the date and time that will be used for subsequent history records added to an NDF (both default history records and those added using NDF_HPUT). If no date and time is set using this routine, then the curent date and time will be used. Any date and time established by a previous call to this function can be removed by supplying a blank value for argument "DATE", in which case the current date and time will be used for subsequent history records.

Invocation:

```
CALL NDF_HSDAT( DATE, INDF, STATUS )
```

Arguments:

DATE = CHARACTER * (*) (Given)
The time and date to be used for subsequent history records, or blank to re-establish the default behaviour (i.e. to use the current time). The allowed formats are described later in the "Date and Time Formats:" section.

INDF = INTEGER (Given)
NDF identifier.

STATUS = INTEGER (Given and Returned)
The global status.

Date and Time Formats :

The formats accepted for the "DATE" argument are listed below. They are all case-insensitive and are generally tolerant of extra white space and alternative field delimiters:

- Gregorian Calendar Date: With the month expressed either as an integer or a 3-character abbreviation, and with optional decimal places to represent a fraction of a day ("1996-10-2" or "1996-Oct-2.6" for example). If no fractional part of a day is given, the time refers to the start of the day (zero hours).
- Gregorian Date and Time: Any calendar date (as above) but with a fraction of a day expressed as hours, minutes and seconds ("1996-Oct-2 12:13:56.985" for example). The date and time can be separated by a space or by a "T" (as used by ISO8601 format).
- Modified Julian Date: With or without decimal places ("MJD 54321.4" for example).
- Julian Date: With or without decimal places ("JD 2454321.9" for example).
- Besselian Epoch: Expressed in decimal years, with or without decimal places ("B1950" or "B1976.13" for example).
- Julian Epoch: Expressed in decimal years, with or without decimal places ("J2000" or "J2100.9" for example).
- Year: Decimal years, with or without decimal places ("1996.8" for example). Such values are interpreted as a Besselian epoch (see above) if less than 1984.0 and as a Julian epoch otherwise.

NDF_HSMOD
Set the history update mode for an NDF

Description:

The routine sets the mode to be used for updating the history component of an NDF. This allows control over the amount of history information subsequently recorded. It also allows history recording to be disabled completely.

Invocation:

```
CALL NDF_HSMOD( HMODE, INDF, STATUS )
```

Arguments:

HMODE = CHARACTER * (*) (Given)

The history update mode required: 'DISABLED', 'QUIET', 'NORMAL' or 'VERBOSE'. This value may be abbreviated, to no less than three characters. In addition, 'SKIP' may be supplied. This is similar to 'DISABLED', in that no history record will be added to the NDF when the NDF is closed. However, 'SKIP' makes no permanent change to the update mode - the next time the NDF is accessed it will have its original update mode.

INDF = INTEGER (Given)

NDF identifier.

STATUS = INTEGER (Given and Returned)

The global status.

NDF_ISACC
Determine whether a specified type of NDF access is available

Description:

The routine determines whether a specified type of access to an NDF is available, or whether it has been disabled. If access is not available, then any attempt to access the NDF in this way will fail.

Invocation:

```
CALL NDF_ISACC( INDF, ACCESS, ISACC, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

ACCESS = CHARACTER * (*) (Given)
The type of NDF access required: 'BOUNDS', 'DELETE', 'SHIFT', 'TYPE' or 'WRITE' (see the Notes section for details).

ISACC = LOGICAL (Returned)
Whether the specified type of access is available.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

The valid access types control the following operations on the NDF:

- 'BOUNDS' permits the pixel-index bounds of a base NDF to be altered.
- 'DELETE' permits deletion of the NDF.
- 'SHIFT' permits pixel-index shifts to be applied to a base NDF.
- 'TYPE' permits the data types of an NDF's components to be altered.
- 'WRITE' permits new values to be written to the NDF, and the state of any of its components to be reset.

NDF_ISBAS
Enquire if an NDF is a base NDF

Description:

The routine returns a logical value indicating whether the NDF whose identifier is supplied is a base NDF (as opposed to an NDF section).

Invocation:

```
CALL NDF_ISBAS( INDF, ISBAS, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

ISBAS = LOGICAL (Returned)
Whether the NDF is a base NDF.

STATUS = INTEGER (Given and Returned)
The global status.

NDF_ISIN
See if one NDF is contained within another NDF

Description:

The routine returns a logical flag indicating if the first supplied NDF is contained within an extension of the second supplied NDF. The search is recursive, so for instance a true value will be returned if the second supplied NDF is contained within an extension of an intermediate NDF that is contained within an extension of the first supplied NDF.

Invocation:

```
CALL NDF_ISIN( INDF1, INDF2, ISIN, STATUS )
```

Arguments:

INDF1 = INTEGER (Given)
The first NDF.

INDF2 = INTEGER (Given)
The second NDF.

ISIN = LOGICAL (Returned)
.TRUE. of the first NDF is contained within an extension of the second NDF, and .FALSE. otherwise.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- A .TRUE. value is returned if the two supplied NDF identifiers refer to the same base NDF.
- If an identifier for an NDF section is supplied to this routine, then the search will be applied to the associated base NDF.
- If this routine is called with STATUS set, then a value of .FALSE. will be returned for the ISIN argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason.

NDF_ISTMP
Enquire if an NDF is temporary

Description:

The routine returns a logical value indicating whether the specified NDF is temporary. Temporary NDFs are deleted once the last identifier which refers to them is annulled.

Invocation:

```
CALL NDF_ISTMP( INDF, ISTMP, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

ISTMP = LOGICAL (Returned)
Whether the NDF is temporary.

STATUS = INTEGER (Given and Returned)
The global status.

NDF_LOC
Obtain an HDS locator for an NDF

Description:

The routine returns an HDS locator for an NDF whose identifier is supplied.

Invocation:

```
CALL NDF_LOC( INDF, MODE, LOC, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

MODE = CHARACTER * (*) (Given)
Mode of access required to the NDF: 'READ', 'UPDATE' or 'WRITE'.

LOC = CHARACTER * (*) (Returned)
HDS locator to the NDF data structure.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If an identifier for an NDF section is supplied to this routine, then the returned locator will refer to the associated base NDF.
- It is the caller's responsibility to annul the locator returned by this routine (by calling the HDS routine DAT_ANNUL) when it is no longer required. The NDF_ system will not perform this task itself.
- If this routine is called with STATUS set, then an invalid locator will be returned for the LOC argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason.
- Although this routine will check the access mode value supplied against the available access to the NDF, HDS does not allow the returned locator to be protected against write access in the case where WRITE access to an NDF is available, but only READ access was requested. In this case it is the responsibility of the caller to respect the locator access restriction.
- The locator returned by this routine should not be used to make alterations to any part of a data structure which is simultaneously being used by the NDF_ system, otherwise there is the possibility of serious internal errors and data corruption.

NDF_MAP
Obtain mapped access to an array component of an NDF

Description:

The routine obtains mapped access to an array component of an NDF, returning a pointer to the mapped values and a count of the number of elements mapped.

Invocation:

```
CALL NDF_MAP( INDF, COMP, TYPE, MMOD, PNTR, EL, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the NDF array component to be mapped: 'DATA', 'QUALITY' or 'VARIANCE' (or 'ERROR').

TYPE = CHARACTER * (*) (Given)
Numeric type to be used for access (e.g. '_REAL').

MMOD = CHARACTER * (*) (Given)
Mapping mode for access to the array: 'READ', 'UPDATE' or 'WRITE', with an optional initialisation mode '/BAD' or '/ZERO' appended.

PNTR(*) = INTEGER (Returned)
Pointer(s) to the mapped values (see the Notes section).

EL = INTEGER (Returned)
Number of elements mapped.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- A comma-separated list of component names may also be given, in which case the routine will map all the requested components using the same data type and mapping mode. Pointers to the values of these mapped components will be returned (in the specified order) in the elements of the array PNTR, which must be of sufficient size to accommodate them.
- The result of mapping the QUALITY component with a data type other than '_UBYTE' is not defined and should not be used.
- If the array is stored in scaled form, then the mapped values will be the result of applying the appropróate scale and zero terms to the elements of the underlying array.
- If the array is stored in delta compressed form, then the mapped values will be the original uncompressed values.
- Scaled and delta arrays are read-only. An error will be reported if the array is stored in scaled or delta form and the access mode is update or write.
- If the MMOD argument includes the '/ZERO' option, the bad pixel flag for the array will be reset to false to indicate that there are no bad values present in the array (see NDF_BAD). If any bad values are then placed into the mapped array, NDF_SBAD should normally be called to set the bad pixel flag true. If this is not done, the bad values in the array may be treated as literal values by subsequent applications.

- If this routine is called with STATUS set, then a value of 1 will be returned for the EL argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason.

NDF_MAPQL
Map the quality component of an NDF as an array of logical values

Description:

The routine maps the quality component of an NDF for read access, returning a pointer to an array of logical values. Elements of this array are set to .TRUE. if the bit-wise "AND" of the corresponding quality value and its effective bad-bits mask gives a zero result, indicating that the corresponding NDF pixel may be used in subsequent processing. Other array elements are set to .FALSE., indicating that corresponding NDF pixels should be excluded from subsequent processing.

Invocation:

```
CALL NDF_MAPQL( INDF, PNTR, EL, BAD, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

PNTR = INTEGER (Returned)
Pointer to the mapped array of logical values.

EL = INTEGER (Returned)
Number of values mapped.

BAD = LOGICAL (Returned)
This argument is set to .TRUE. if any of the mapped values is set to .FALSE. (i.e. if any NDF pixels are to be excluded as a consequence of the associated quality values). Otherwise it is set to .FALSE..

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If the quality component's values are undefined, then this routine will return a pointer to an array of .TRUE. values.
- Note that this routine only obtains read access to the quality component; changes made to the mapped values will not be reflected in changes to the NDF's quality values.
- This routine disables automatic quality masking, so that subsequent access to other NDF array components via the same identifier will take no account of the possible presence of associated quality values.
- If this routine is called with STATUS set, then a value of 1 will be returned for the EL argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason.

NDF_MAPZ
Obtain complex mapped access to an array component of an NDF

Description:

The routine obtains complex mapped access to an array component of an NDF, returning pointers to the mapped real and imaginary values and a count of the number of elements mapped.

Invocation:

```
CALL NDF_MAPZ( INDF, COMP, TYPE, MMOD, RPNTR, IPNTR, EL, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the NDF array component to be mapped: 'DATA' or 'VARIANCE' (or 'ERROR').

TYPE = CHARACTER * (*) (Given)
Numeric type to be used for access (e.g. '_REAL').

MMOD = CHARACTER * (*) (Given)
Mapping mode for access to the array: 'READ', 'UPDATE' or 'WRITE', with an optional initialisation mode '/ZERO' or '/BAD' appended.

RPNTR(*) = INTEGER (Returned)
Pointer(s) to the mapped real (i.e. non-imaginary) values (see the Notes section).

IPNTR(*) = INTEGER (Returned)
Pointer(s) to the mapped imaginary values (see the Notes section).

EL = INTEGER (Returned)
Number of elements mapped.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- A comma-separated list of component names may also be given, in which case the routine will map all the requested components using the same data type and mapping mode. Pointers to the values of these mapped components will be returned (in the specified order) in the elements of the arrays RPNTR and IPNTR, which must be of sufficient size to accommodate them.
- Access to an NDF's QUALITY component is not available using this routine.
- If this routine is called with STATUS set, then a value of 1 will be returned for the EL argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason.

NDF_MBAD
Merge the bad-pixel flags of the array components of a pair of NDFs

Description:

The routine merges the bad-pixel flag values of an array component (or components) for a pair of NDFs, returning the logical "OR" of the separate values for each NDF. In addition, if bad pixels are found to be present in either NDF but the application indicates that it cannot correctly handle such values, then an error to this effect is reported and a STATUS value is set.

Invocation:

```
CALL NDF_MBAD( BADOK, INDF1, INDF2, COMP, CHECK, BAD, STATUS )
```

Arguments:

BADOK = LOGICAL (Given)
Whether the application can correctly handle NDF array components containing bad pixel values.

INDF1 = INTEGER (Given)
Identifier for the first NDF whose bad-pixel flag value is to be merged.

INDF2 = INTEGER (Given)
Identifier for the second NDF.

COMP = CHARACTER * (*) (Given)
Name of the NDF array component: 'DATA', 'QUALITY' or 'VARIANCE'.

CHECK = LOGICAL (Given)
Whether to perform explicit checks to see whether bad pixels are actually present. (This argument performs the same function as in the routine NDF_BAD.)

BAD = LOGICAL (Returned)
The combined bad-pixel flag value (the logical "OR" of the values obtained for each NDF).

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- A comma-separated list of component names may also be supplied, in which case the routine will take the logical "OR" of all the specified components when calculating the combined bad-pixel flag value.
- The effective value of the bad-pixel flag for each NDF array component which this routine uses is the same as would be returned by a call to the routine NDF_BAD.
- If this routine detects the presence of bad pixels which the application cannot support (as indicated by a .FALSE. value for the BADOK argument), then an error will be reported to this effect and a STATUS value of NDF__BADNS (bad pixels not supported) will be returned. The value of the BAD argument will be set to .TRUE. under these circumstances. The NDF__BADNS constant is defined in the include file NDF_ERR.

NDF_MBADN
Merge the bad-pixel flags of the array components of a number of NDFs

Description:

The routine merges the bad-pixel flag values of an array component (or components) for a number of NDFs, returning the logical "OR" of the separate values for each NDF. In addition, if bad pixels are found to be present in any NDF but the application indicates that it cannot correctly handle such values, then an error to this effect is reported and a STATUS value is set.

Invocation:

```
CALL NDF_MBADN( BADOK, N, NDFS, COMP, CHECK, BAD, STATUS )
```

Arguments:

BADOK = LOGICAL (Given)
Whether the application can correctly handle NDF array components containing bad pixel values.

N = INTEGER (Given)
Number of NDFs whose bad-pixel flags are to be merged.

NDFS(N) = INTEGER (Given)
Array of identifiers for the NDFs to be merged.

COMP = CHARACTER * (*) (Given)
Name of the NDF array component: 'DATA', 'QUALITY' or 'VARIANCE'.

CHECK = LOGICAL (Given)
Whether to perform explicit checks to see whether bad pixels are actually present. (This argument performs the same function as in the routine NDF_BAD.)

BAD = LOGICAL (Returned)
The combined bad-pixel flag value (the logical "OR" of the values obtained for each NDF).

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- A comma-separated list of component names may also be supplied, in which case the routine will take the logical "OR" of all the specified components when calculating the combined bad-pixel flag value.
- The effective value of the bad-pixel flag for each NDF array component which this routine uses is the same as would be returned by a call to the routine NDF_BAD.
- If this routine detects the presence of bad pixels which the application cannot support (as indicated by a .FALSE. value for the BADOK argument), then an error will be reported to this effect and a STATUS value of NDF__BADNS (bad pixels not supported) will be returned. The value of the BAD argument will be set to .TRUE. under these circumstances. The NDF__BADNS constant is defined in the include file NDF_ERR.

NDF_MBND
Match the pixel-index bounds of a pair of NDFs

Description:

The routine matches the pixel-index bounds of a pair of NDFs so that their array components may be compared pixel-for-pixel during subsequent processing. Matching is performed by selecting an appropriate section from each NDF, the method used to define this section being determined by the value given for the OPTION argument.

Invocation:

```
CALL NDF_MBND( OPTION, INDF1, INDF2, STATUS )
```

Arguments:

OPTION = CHARACTER * (*) (Given)
This argument determines how the section to be selected from each NDF is defined: 'PAD' or 'TRIM' (see the Notes section for details). Its value may be abbreviated to 3 characters.

INDF1 = INTEGER (Given and Returned)
Identifier for the first NDF whose pixel-index bounds are to be matched.

INDF2 = INTEGER (Given and Returned)
Identifier for the second NDF to be matched.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If OPTION='PAD' is specified, then the NDF bounds will be matched by "padding"; i.e. each NDF will be extended by selecting the smallest section from it which encompasses all the pixels in both NDFs. In effect, the pixel-index bounds of the two NDFs are "maximised" and the "union" of the two sets of pixels is selected. Any new pixels introduced into either NDF will be padded with the "bad" value. If the NDFs have different numbers of dimensions, then the dimensionality of both the returned sections will match the NDF with the higher dimensionality.
- If OPTION='TRIM' is specified, then the NDF bounds will be matched by "trimming"; i.e. each NDF will be restricted in extent by selecting a section from it which encompasses only those pixels which are present in both NDFs. In effect, the pixel-index bounds of the two NDFs are "minimised" and the "intersection" of the two sets of pixels is selected. An error will result if the two NDFs have no pixels in common. If the NDFs have different numbers of dimensions, then the dimensionality of both the returned sections will match the NDF with the lower dimensionality.
- Note that the initial NDF identifier values will be annulled by this routine and replaced with identifiers describing appropriate new sections from the original NDFs. If access to the original data is still required, then the initial identifiers may be cloned with the routine NDF_CLONE before calling this routine.

NDF_MBNDN
Match the pixel-index bounds of a number of NDFs

Description:

The routine matches the pixel-index bounds of a number of NDFs so that their array components may be compared pixel-for-pixel during subsequent processing. Matching is performed by selecting an appropriate section from each NDF, the method used to define this section being determined by the value given for the OPTION argument.

Invocation:

```
CALL NDF_MBNDN( OPTION, N, NDFS, STATUS )
```

Arguments:

OPTION = CHARACTER * (*) (Given)
This argument determines how the section to be selected from each NDF is defined: 'PAD' or 'TRIM' (see the Notes section for details). Its value may be abbreviated to 3 characters.

N = INTEGER (Given)
Number of NDFs whose pixel-index bounds are to be matched.

NDFS(N) = INTEGER (Given and Returned)
Array of identifiers for the NDFs to be matched.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If OPTION='PAD' is specified, then the NDF bounds will be matched by "padding"; i.e. each NDF will be extended by selecting the smallest section from it which encompasses all the pixels in all the NDFs. In effect, the pixel-index bounds of the NDFs are "maximised" and the "union" of all N sets of pixels is selected. Any new pixels introduced into an NDF will be padded with the "bad" value. If the NDFs have different numbers of dimensions, then the dimensionality of all the returned sections will match the NDF with the highest dimensionality.
- If OPTION='TRIM' is specified, then the NDF bounds will be matched by "trimming"; i.e. each NDF will be restricted in extent by selecting a section from it which encompasses only those pixels which are present in all the NDFs. In effect, the pixel-index bounds of the NDFs are "minimised" and the "intersection" of all N sets of pixels is selected. An error will result if the NDFs have no pixels in common. If the NDFs have different numbers of dimensions, then the dimensionality of all the returned sections will match the NDF with the lowest dimensionality.
- Note that the initial NDF identifier values will be annulled by this routine and replaced with identifiers describing appropriate new sections from the original NDFs. If access to the original data is still required, then the initial identifiers may be cloned with the routine NDF_CLONE before calling this routine.

NDF_MSG
Assign the name of an NDF to a message token

Description:

The routine assigns the name of an NDF to a message token (in a form which a user will understand) for use in constructing messages with the ERR_ and MSG_ routines (see SUN/104).

Invocation:

```
CALL NDF_MSG( TOKEN, INDF )
```

Arguments:

TOKEN = CHARACTER * (*) (Given)
Name of the message token.

INDF = INTEGER (Given)
NDF identifier.

Notes:

- This routine has no STATUS argument and performs no error checking. If it should fail, then no assignment to the message token will be made and this will be apparent in the final message.

NDF_MTYPE
Match the types of the array components of a pair of NDFs

Description:

The routine matches the types of the array components of a pair of NDFs, selecting a numeric type which an application may use to process these components. It also returns the type which should be used for storing the result of this processing.

Invocation:

```
CALL NDF_MTYPE( TYPLST, INDF1, INDF2, COMP, ITYPE, DTYPE, STATUS )
```

Arguments:

TYPLST = CHARACTER * (*) (Given)
A comma-separated list of the numeric types which the application can process explicitly; e.g. '_INTEGER,_REAL'. The first type which has sufficient precision will be selected from this list, so they should normally be given in order of increasing computational cost.

INDF1 = INTEGER (Given)
Identifier for the first NDF whose type is to be matched.

INDF2 = INTEGER (Given)
Identifier for the second NDF.

COMP = CHARACTER * (*) (Given)
Name of the NDF array component whose type is to be considered.

ITYPE = CHARACTER * (*) (Returned)
Numeric type which the application should use to process the NDF components. This value is returned as an upper case character string of maximum length NDF__SZTYP. Its value is the first entry in the TYPLST list to which the NDF array components may be converted without unnecessary loss of information.

DTYPE = CHARACTER * (*) (Returned)
Data type required to hold the result of processing the NDF array components. This result is returned as an upper case character string of maximum length NDF__SZFTP. It is intended to be used as input to the NDF_STYPE routine to set the type of the output NDF component into which the result will be written.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- A comma-separated list of component names may also be supplied, in which case the results returned by this routine will take account of the types of all the specified components in both NDFs.
- Matching of the type of a single NDF to an application may be performed by supplying the same identifier value for both the INDF1 and INDF2 arguments. There is no extra cost in doing this.
- If the TYPLST argument does not specify any type to which the NDF components may be converted without loss of information, then the routine will return the highest precision type which is available. An error will be reported, however, and STATUS will be set to NDF__TYPNI (type not implemented).

- The constants NDF__SZTYP and NDF__SZFTP are defined in the include file NDF_PAR. The error code NDF__TYPNI is defined in the include file NDF_ERR.

NDF_MTYPN
Match the types of the array components of a number of NDFs

Description:

The routine matches the types of the array components of a number of NDFs, selecting a type which an application may use to process these components. It also returns the numeric type which should be used for storing the result of this processing.

Invocation:

```
CALL NDF_MTYPN( TYPLST, N, NDFS, COMP, ITYPE, DTYPE, STATUS )
```

Arguments:

TYPLST = CHARACTER * (*) (Given)
A comma-separated list of the numeric types which the application can process explicitly; e.g. '_INTEGER,_REAL'. The first type which has sufficient precision will be selected from this list, so they should normally be given in order of increasing computational cost.

N = INTEGER (Given)
Number of NDFs whose types are to be matched.

NDFS(N) = INTEGER (Given)
Array of identifiers for the NDFs to be matched.

COMP = CHARACTER * (*) (Given)
Name of the NDF array component whose type is to be considered.

ITYPE = CHARACTER * (*) (Returned)
Numeric type which the application should use to process the NDF components. This value is returned as an upper case character string of maximum length NDF__SZTYP. Its value is the first entry in the TYPLST list to which the NDF array components may be converted without unnecessary loss of information.

DTYPE = CHARACTER * (*) (Returned)
Data type required to hold the result of processing the NDF array components. This result is returned as an upper case character string of maximum length NDF__SZFTP. It is intended to be used as input to the NDF_STYPE routine to set the type of the output NDF component into which the result will be written.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- A comma-separated list of component names may also be supplied, in which case the results returned by this routine will take account of the types of all the specified components in all the NDFs.
- If the TYPLST argument does not specify any type to which the NDF components may be converted without loss of information, then the routine will return the highest precision type which is available. An error will be reported, however, and STATUS will be set to NDF__TYPNI (type not implemented).
- The constants NDF__SZTYP and NDF__SZFTP are defined in the include file NDF_PAR. The error code NDF__TYPNI is defined in the include file NDF_ERR.

NDF_NBLOC
Determine the number of blocks of adjacent pixels in an NDF

Description:

The routine determines the number of "blocks" (i.e. sections) of adjacent pixels that can be obtained from an NDF, subject to the constraint that no block should exceed a specified maximum number of pixels in any dimension. More specifically, given the maximum size in pixels of a block in each dimension (MXDIM), this routine returns the maximum value which can be supplied for the IBLOCK argument of the routine NDF_BLOCK if a valid NDF identifier for a block of adjacent pixels is to be returned.

Invocation:

```
CALL NDF_NBLOC( INDF, NDIM, MXDIM, NBLOCK, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

NDIM = INTEGER (Given)
Number of maximum dimension sizes.

MXDIM(NDIM) = INTEGER (Given)
Array specifying the maximum size of a block in pixels along each dimension.

NBLOCK = INTEGER (Returned)
Number of blocks which can be obtained from the NDF.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- This routine is provided to calculate an upper bound on the number of blocks for DO-loops which process NDFs by dividing them into separate blocks by means of calls to the routine NDF_BLOCK.
- If the number of maximum dimension sizes supplied (NDIM) is less than the number of NDF dimensions, then a value of 1 will be used for the extra dimension sizes. If the value of NDIM is larger than this number, then the excess dimension sizes will be ignored.
- A value of zero will be returned for the NBLOCK argument if this routine is called with STATUS set. The same value will also be returned if the routine should fail for any reason.

NDF_NCHNK
Determine the number of chunks of contiguous pixels in an NDF

Description:

The routine determines the number of "chunks" (i.e. sections) of contiguous pixels that can be obtained from an NDF, subject to the constraint that no chunk should contain more than a specified maximum number of pixels. More specifically, given the maximum number of pixels in a chunk (MXPIX), this routine returns the maximum value which can be supplied for the ICHUNK argument of the routine NDF_CHUNK if a valid NDF identifier for a chunk of contiguous pixels is to be returned.

Invocation:

```
CALL NDF_NCHNK( INDF, MXPIX, NCHUNK, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

MXPIX = INTEGER (Given)
Maximum number of contiguous pixels required in each chunk.

NCHUNK = INTEGER (Returned)
Number of chunks which can be obtained from the NDF.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- This routine is provided to calculate an upper bound on the number of chunks for DO-loops which process NDFs by dividing them into separate chunks by means of calls to the routine NDF_CHUNK.
- A value of zero will be returned for the NCHUNK argument if this routine is called with STATUS set. The same value will also be returned if the routine should fail for any reason.

NDF_NEW
Create a new simple NDF

Description:

The routine creates a new simple NDF and returns an identifier for it. The NDF may subsequently be manipulated with the NDF_ routines.

Invocation:

```
CALL NDF_NEW( FTYPE, NDIM, LBND, UBND, PLACE, INDF, STATUS )
```

Arguments:

FTYPE = CHARACTER ∗ (∗) (Given)
Full type of the NDF's DATA component (e.g. '_REAL' or 'COMPLEX_INTEGER').

NDIM = INTEGER (Given)
Number of NDF dimensions.

LBND(NDIM) = INTEGER (Given)
Lower pixel-index bounds of the NDF.

UBND(NDIM) = INTEGER (Given)
Upper pixel-index bounds of the NDF.

PLACE = INTEGER (Given and Returned)
An NDF placeholder (e.g. generated by the NDF_PLACE routine) which indicates the position in the data system where the new NDF will reside. The placeholder is annulled by this routine, and a value of NDF__NOPL will be returned (as defined in the include file NDF_PAR).

INDF = INTEGER (Returned)
Identifier for the new NDF.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- This routine creates a "simple" NDF, i.e. one whose array components will be stored in "simple" form by default (see SGP/38).
- The full data type of the DATA component is specified via the FTYPE argument and the data type of the VARIANCE component defaults to the same value. These data types may be set individually with the NDF_STYPE routine if required.
- If this routine is called with STATUS set, then a value of NDF__NOID will be returned for the INDF argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. In either event, the placeholder will still be annulled. The NDF__NOID constant is defined in the include file NDF_PAR.

NDF_NEWP
Create a new primitive NDF

Description:

The routine creates a new primitive NDF and returns an identifier for it. The NDF may subsequently be manipulated with the NDF_ routines.

Invocation:

```
CALL NDF_NEWP( FTYPE, NDIM, UBND, PLACE, INDF, STATUS )
```

Arguments:

FTYPE = CHARACTER ∗ (∗) (Given)
Data type of the NDF's DATA component (e.g. '_REAL'). Note that complex types are not permitted when creating a primitive NDF.

NDIM = INTEGER (Given)
Number of NDF dimensions.

UBND(NDIM) = INTEGER (Given)
Upper pixel-index bounds of the NDF (the lower bound of each dimension is taken to be 1).

PLACE = INTEGER (Given and Returned)
An NDF placeholder (e.g. generated by the NDF_PLACE routine) which indicates the position in the data system where the new NDF will reside. The placeholder is annulled by this routine, and a value of NDF__NOPL will be returned (as defined in the include file NDF_PAR).

INDF = INTEGER (Returned)
Identifier for the new NDF.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- This routine creates a "primitive" NDF, i.e. one whose array components are stored in "primitive" form by default (see SGP/38).
- The full type of the DATA component is specified via the FTYPE argument and the type of the VARIANCE component defaults to the same value. These types may be set individually with the NDF_STYPE routine if required.
- If this routine is called with STATUS set, then a value of NDF__NOID will be returned for the INDF argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. In either event, the placeholder will still be annulled. The NDF__NOID constant is defined in the include file NDF_PAR.

NDF_NOACC
Disable a specified type of access to an NDF

Description:

The routine disables the specified type of access to an NDF, so that any subsequent attempt to access it in that way will fail. Access restrictions imposed on an NDF identifier by this routine will be propagated to any new identifiers derived from it, and cannot be revoked.

Invocation:

```
CALL NDF_NOACC( ACCESS, INDF, STATUS )
```

Arguments:

ACCESS = CHARACTER * (*) (Given)
The type of access to be disabled: 'BOUNDS', 'DELETE', 'MODIFY', 'SHIFT', 'TYPE' or 'WRITE'.

INDF = INTEGER (Given)
NDF identifier.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

Disabling each type of access imposes the following restrictions on an NDF:

- 'BOUNDS' prevents the pixel-index bounds of a base NDF from being altered.
- 'DELETE' prevents an NDF from being deleted.
- 'MODIFY' prevents any form of modification to the NDF (i.e. it disables all the other access types).
- 'SHIFT' prevents pixel-index shifts from being applied to a base NDF.
- 'TYPE' prevents the data type of any NDF components from being altered.
- 'WRITE' prevents new values from being written to the NDF, or the state of any of its components from being reset.

NDF_OPEN
Open an existing or new NDF

Description:

The routine finds an existing NDF data structure and returns an identifier for it, or creates a placeholder for a new NDF.

Invocation:

```
CALL NDF_OPEN( LOC, NAME, MODE, STAT, INDF, PLACE, STATUS )
```

Arguments:

LOC = CHARACTER * (*) (Given)
Locator to the enclosing HDS structure.

NAME = CHARACTER * (*) (Given)
Name of the HDS structure component.

MODE = CHARACTER * (*) (Given)
Type of NDF access required: 'READ', 'UPDATE' or 'WRITE'.

STAT = CHARACTER * (*) (Given)
The state of the NDF, specifying whether it is known to exist or not: 'NEW', 'OLD', or 'UNKNOWN'.

INDF = INTEGER (Returned)
NDF identifier.

PLACE = INTEGER (Returned)
NDF placeholder identifying the nominated position in the data system.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If the STAT argument is set to 'NEW', then this routine will return a placeholder for a new NDF. If STAT is set to 'OLD', it will search for an existing NDF. If STAT is set to 'UNKNOWN', it will first search for an existing NDF but will return a placeholder for a new NDF if an existing one cannot be found.
- If this routine succeeds, then a valid value will be returned for INDF if the NDF already existed, or for PLACE if it did not exist. The unused return argument will be set to the appropriate null value (NDF__NOID or NDF__NOPL respectively).
- If 'WRITE' access is specified for an existing NDF, then all the NDF's components will be reset to an undefined state ready to receive new values. If 'UPDATE' access is specified, the NDF's components will retain their values, which may then be modified.
- An error will result if the STAT argument is set to 'OLD' but no existing NDF could be found. An error will also result if a placeholder for a new NDF is to be returned but 'READ' access was requested.
- The value given for the NAME argument may be an HDS path name, consisting of several fields separated by '.', so that an NDF can be opened in a sub-component (or a sub-sub-component...) of the structure identified by the locator LOC. Array subscripts may also be used in this component name. Thus a string such as 'MYSTRUC.ZONE(2).IMAGE' could be used as a valid NAME value.

- An NDF can be opened within an explicitly named container file by supplying the symbolic value DAT__ROOT for the LOC argument and giving a full HDS object name (including a container file specification) for the NAME argument.
- If a blank value is given for the NAME argument, then the NDF will be the object identified directly by the locator LOC.
- If a placeholder is to be returned and the new NDF is to be a top-level object, then a new container file will be created. Otherwise, the container file and all structures lying above the new NDF should already exist.
- If the LOC and NAME arguments identify a pre-existing object which is not a top-level object, then this may be used as the basis for the new NDF. An object which is to be used in this way must be an empty scalar structure with an HDS type of 'NDF'.
- The locator supplied as input to this routine may later be annulled without affecting the behaviour of the NDF_ system.
- If this routine is called with STATUS set, then a value of NDF__NOPL will be returned for the PLACE argument, and a value of NDF__NOID will be returned for the INDF argument, although no further processing will occur. The same values will also be returned if the routine should fail for any reason.
- The NDF__NOPL and NDF__NOID constants are defined in the include file NDF_PAR. The DAT__ROOT constant is defined in the include file DAT_PAR (see SUN/92).

NDF_PLACE
Obtain an NDF placeholder

Description:

The routine returns an NDF placeholder. A placeholder is used to identify a position in the underlying data system (HDS) and may be passed to other routines (e.g. NDF_NEW) to indicate where a newly created NDF should be positioned.

Invocation:

```
CALL NDF_PLACE( LOC, NAME, PLACE, STATUS )
```

Arguments:

LOC = CHARACTER * (*) (Given)
HDS locator to the structure to contain the new NDF.

NAME = CHARACTER * (*) (Given)
Name of the new structure component (i.e. the NDF).

PLACE = INTEGER (Returned)
NDF placeholder identifying the nominated position in the data system.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- Placeholders are intended only for local use within an application and only a limited number of them are available simultaneously. They are always annulled as soon as they are passed to another routine to create a new NDF, where they are effectively exchanged for an NDF identifier.
- The value given for the NAME argument may be an HDS path name, consisting of several fields separated by '.', so that an NDF can be created in a sub-component (or a sub-sub-component...) of the structure identified by the locator LOC. Array subscripts may also be used in this component name. Thus a string such as 'MYSTRUC.ZONE(2).IMAGE' could be used as a valid NAME value.
- Normally, this routine will be used as the basis for creating a completely new NDF data structure. However, if the LOC and NAME arguments refer to a pre-existing object, then this structure will be used as the basis for the new NDF. An object which is to be used in this way must be an empty scalar structure with an HDS type of 'NDF'.
- A new NDF can be created within an explicitly named container file by supplying the symbolic value DAT__ROOT for the LOC argument, and specifying the container file within the value supplied for the NAME argument. If the object is the top level object within a container file, then a new container file is created. If it is not a top level object, then the container file and all structures lying above the object should already exist.
- If a blank value is given for the NAME argument, then the new NDF will be the object identified directly by the locator LOC. This must be an empty scalar structure of type 'NDF'.
- If this routine is called with STATUS set, then a value of NDF__NOPL will be returned for the PLACE argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason.
- The NDF__NOPL constant is defined in the include file NDF_PAR. The DAT__ROOT constant is defined in the include file DAT_PAR (see SUN/92).

NDF_PROP
Propagate NDF information to create a new NDF via the ADAM parameter system

Description:

The routine creates a new NDF data structure through the ADAM parameter system, associates it with a parameter and returns an identifier for it. The shape, data type, etc. of this new NDF are based on a existing "template" NDF, and the values of components of this template may be selectively propagated to initialise the new data structure.

Invocation:

```
CALL NDF_PROP( INDF1, CLIST, PARAM, INDF2, STATUS )
```

Arguments:

INDF1 = INTEGER (Given)
Identifier for an existing NDF (or NDF section) to act as a template.

CLIST = CHARACTER * (*) (Given)
A comma-separated list of the NDF components which are to be propagated to the new data structure. By default, the HISTORY, LABEL and TITLE components are propagated. All extensions are also propagated by default except for any that have had a zero value assigned to the corresponding "PXT..." tuning parameter using NDF_TUNE. See the "Component Propagation" section for further details.

PARAM = CHARACTER * (*) (Given)
Name of the ADAM parameter for the new NDF.

INDF2 = INTEGER (Returned)
Identifier for the new NDF.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If this routine is called with STATUS set, then a value of NDF__NOID will be returned for the INDF2 argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The NDF__NOID constant is defined in the include file NDF_PAR.

Component Propagation :

- The template components whose values are to be propagated to initialise the new data structure are specified via the CLIST argument. Thus CLIST='DATA,QUALITY' would cause the new NDF to inherit its DATA and QUALITY values (if available) from the template structure, in addition to those propagated by default. Component propagation may be suppressed by supplying a component name with the prefix 'NO'. Thus CLIST='DATA,NOHISTORY' would propagate the DATA component, but suppress propagation of HISTORY. If component names appear more than once in the CLIST value, then the last occurrence takes precedence.

- Propagation of specific NDF extensions may be suppressed by using 'NOEXTENSION()' as one of the items in the CLIST argument; a list of the extensions to be suppressed should appear between the parentheses. Thus CLIST='AXIS,NOEXTENSION(IRAS,ASTERIX)' would propagate the AXIS component, but suppress propagation of the IRAS and ASTERIX extensions (if present). Propagation of suppressed extensions may be re-enabled by specifying 'EXTENSION()' in a similar manner at a later point in the CLIST value.
- An asterisk (∗) may be used as a wild card to match all extension names. Thus 'NOEXTENSION(∗),EXTENSION(may be used to indicate that only the IRAS extension should be propagated.
- Whether or not a named extension is propagated by default can be controlled via an NDF tuning parameter (see NDF_TUNE). The defaults established using NDF_TUNE can be over-ridden by specifying the extension explicitly within the CLIST parameter; e.g. 'EXTENSION(FITS)' or 'NOEXTENSION(FITS)' can be used to over-ride the default established by the PXTFITS tuning parameter.
- Component names in the CLIST argument may be abbreviated to 3 characters, but extension names must appear in full.

NDF_PTSZx
Set new scale and zero values for an NDF array component

Description:

The routine sets new values for the scale and zero values associated with an NDF arraycomponent. If the array is stored in simple or primitive form, then the storage form is changed to scaled. See also NDF_ZSCAL which provides a higher level interface for creating SCALED arrays.

Invocation:

```
CALL NDF_PTSZx( SCALE, ZERO, INDF, COMP, STATUS )
```

Arguments:

SCALE = ? (Given)
The new value for the scaling factor.

ZERO = ? (Given)
The new value for the zero offset.

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the NDF array component; 'DATA' or 'VARIANCE'.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- There is a routine for each of the standard Fortran numerical data types: integer, real and double precision. Replace the (lower case) "x" in the routine name by I, R or D as appropriate.
- A comma-separated list of component names may also be supplied, in which case the same scale and zero values will be used for each component in turn.
- This routine may only be used to change the scaling of a base NDF. If it is called with an array which is not a base array, then it will return without action. No error will result.
- An error will result if the array component, or any part of it, is currently mapped for access (e.g. through another identifier).
- This routine has no effect on components which are in an undefined state.

NDF_PTWCS
Store world coordinate system information in an NDF

Description:

The routine stores new world coordinate system (WCS) information in an NDF, over-writing any already present.

Invocation:

```
CALL NDF_PTWCS( IWCS, INDF, STATUS )
```

Arguments:

IWCS = INTEGER (Given)
An AST pointer to a FrameSet (SUN/210) containing information about the new world coordinate systems to be associated with the NDF.

INDF = INTEGER (Given)
NDF identifier.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- This routine may only be used to store WCS information in a base NDF. If an NDF section is supplied, it will simply return without action.
- The AST FrameSet supplied must conform to the various restrictions imposed by the NDF_ system (e.g. on the nature of its base Frame). An error will result if any of these restrictions is not met.
- This routine makes a copy of the information in the FrameSet supplied, so the original FrameSet may subsequently be modified without affecting the behaviour of the NDF_ system.

NDF_QMASK
Combine an NDF quality value with a bad-bits mask to give a logical result

Description:

This function may be used to combine an NDF quality array value with the associated bad-bits mask value to derive a logical result indicating whether an NDF pixel should be included or excluded from processing by general-purpose software.

Invocation:

```
RESULT = NDF_QMASK( QUAL, BADBIT )
```

Arguments:

QUAL = BYTE (Given)
The unsigned byte quality value.

BADBIT = BYTE (Given)
The unsigned byte bad-bits mask value.

Returned Value:

NDF_QMASK = LOGICAL
If the function returns a .TRUE. result, then the pixel with quality value QUAL should be included in processing by general-purpose software. If it returns a .FALSE. result, then the pixel should be regarded as "bad" and excluded from processing.

Notes:

- This function is implemented as a Fortran statement function and should be defined in each program unit from which it is invoked by means of the include file NDF_FUNC. This file should normally be included immediately after any local variable declarations.
- The result of this function is computed by forming the bit-wise "AND" between the QUAL and BADBIT values and testing the result for equality with zero. Its actual implementation is machine-dependent.

NDF_QMF
Obtain the logical value of an NDF's quality masking flag

Description:

The routine returns the current value of an NDF's logical quality masking flag. This flag determines whether the NDF's quality component (if present) will be used to generate "bad" pixel values for automatic insertion into the data and variance arrays when these are accessed in READ or UPDATE mode. Normally, this automatic quality masking is used to convert quality information into "bad" pixels so that an application need not consider the quality information explicitly. If the quality masking flag is set to .FALSE., then automatic masking will not occur so that the application can process the quality component by accessing it directly.

Invocation:

```
CALL NDF_QMF( INDF, QMF, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

QMF = LOGICAL (Returned)
The value of the quality masking flag.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- A quality masking flag is associated with each NDF identifier and is initially set to .TRUE.. Its value changes to .FALSE. whenever the quality component is accessed directly (e.g. using NDF_MAP or NDF_MAPQL) and reverts to .TRUE. when access is relinquished (e.g. using NDF_UNMAP). This default behaviour may also be over-ridden by calling NDF_SQMF to set its value explicitly. NDF_QMF allows the current value to be determined.

NDF_RESET
Reset an NDF component to an undefined state

Description:

The routine resets a component of an NDF so that its value becomes undefined. It may be used to remove unwanted optional NDF components. Its use is also advisable before making format changes to an NDF if retention of the existing values is not required (e.g. before changing the data type of an array component with the NDF_STYPE routine); this will avoid the cost of converting the existing values.

Invocation:

```
CALL NDF_RESET( INDF, COMP, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the NDF component to be reset; any NDF component name is valid. No error will result if the component is already undefined.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- A comma-separated list of component names may also be supplied in which case each component will be reset in turn.
- Specifying a component name of '*' will cause all components, except for HISTORY and extensions, to be reset. The former may be reset by specifying its name explicitly, while all extensions may be removed by specifying a component name of 'EXTENSION'.
- Individual extensions may be removed from an NDF with the NDF_XDEL routine.
- This routine may only be used to reset components of a base NDF. If an NDF section is supplied, then it will return without action. No error will result.
- An array component of an NDF cannot be reset while it is mapped for access. Neither can an NDF's axis component be reset while any axis array is mapped for access. This routine will fail if either of these conditions occurs.

NDF_SAME
Enquire if two NDFs are part of the same base NDF

Description:

The routine determines whether two NDF identifiers refer to parts of the same base NDF. If so, it also determines whether their transfer windows intersect.

Invocation:

```
CALL NDF_SAME( INDF1, INDF2, SAME, ISECT, STATUS )
```

Arguments:

INDF1 = INTEGER (Given)
Identifier for the first NDF (or NDF section).

INDF2 = INTEGER (Given)
Identifier for the second NDF (or NDF section).

SAME = LOGICAL (Returned)
Whether the identifiers refer to parts of the same base NDF.

ISECT = LOGICAL (Returned)
Whether their transfer windows intersect.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If the transfer windows of the two NDFs (or NDF sections) intersect, then (i) they both refer to the same base NDF, and (ii) altering values in an array component of one of the NDFs can result in the values in the corresponding component of the other NDF changing in consequence. Thus, the array components of the two NDFs are not mutually independent.

NDF_SBAD
Set the bad-pixel flag for an NDF array component

Description:

The routine sets the value of the bad-pixel flag for an NDF array component. A call to this routine with BAD set to .TRUE. declares that the specified component may contain bad pixel values for which checks must be made by algorithms which subsequently process its values. A call with BAD set to .FALSE. declares that there are definitely no bad values present and that subsequent checks for such values may be omitted.

Invocation:

```
CALL NDF_SBAD( BAD, INDF, COMP, STATUS )
```

Arguments:

BAD = LOGICAL (Given)
Bad-pixel flag value to be set.

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the NDF array component; 'DATA' or 'VARIANCE'.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- A comma-separated list of component names may also be supplied, in which case the bad-pixel flag will be set to the same value for each component in turn.
- If a component is mapped for access when this routine is called, then the bad-pixel flag will be associated with the mapped values. This information will only be transferred to the actual data object when the component is unmapped (but only if it was mapped for UPDATE or WRITE access). The value transferred may be modified if conversion errors occur during the unmapping process.
- This routine has no effect on components which are in an undefined state; the bad-pixel flag for such components always remains set to .TRUE. (or .FALSE. in the case of the QUALITY component).

NDF_SBB
Set a bad-bits mask value for the quality component of an NDF

Description:

The routine assigns a new unsigned byte bad-bits mask value to the quality component of an NDF.

Invocation:

```
CALL NDF_SBB( BADBIT, INDF, STATUS )
```

Arguments:

BADBIT = BYTE (Given)
The unsigned byte bad-bits mask value.

INDF = INTEGER (Given)
NDF identifier.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If WRITE access to the NDF is not available, or if an NDF section is supplied (as opposed to a base NDF), then no permanent change to the data object will be made. In this case, the new bad-bits value will be associated with the NDF identifier and will subsequently be used by other NDF_ routines which access the NDF through this identifier. The new value will also be propagated to any new identifiers derived from it.

NDF_SBND
Set new pixel-index bounds for an NDF

Description:

The routine sets new pixel-index bounds for an NDF (or NDF section). The number of NDF dimensions may also be changed. If a base NDF is specified, then a permanent change is made to the actual data object and this will be apparent through any other NDF identifiers which refer to it. However, if an identifier for an NDF section is specified, then its bounds are altered without affecting other identifiers.

Invocation:

```
CALL NDF_SBND( NDIM, LBND, UBND, INDF, STATUS )
```

Arguments:

NDIM = INTEGER (Given)
New number of NDF dimensions.

LBND(NDIM) = INTEGER (Given)
New lower pixel-index bounds of the NDF.

UBND(NDIM) = INTEGER (Given)
New upper pixel-index bounds of the NDF.

INDF = INTEGER (Given)
NDF identifier.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- The bounds of an NDF section cannot be altered while any of its array components (or any of its axis arrays) is mapped for access through the identifier supplied to this routine.
- The bounds of a base NDF cannot be altered while any part of any of its array components (or any of its axis arrays) is mapped for access, even through another identifier.
- The pixel values of any defined NDF array component will be retained if those pixels lie within both the old and new bounds. Any pixels lying outside the new bounds will be lost and cannot later be recovered by further changes to the NDF's bounds. Any new pixels introduced where the new bounds extend beyond the old ones will be assigned the "bad" value, and subsequent enquiries about the presence of bad pixels will reflect this.
- If the new NDF bounds extend beyond the bounds of the associated base NDF and any of the NDF's axis arrays have defined values, then these values will be extrapolated as necessary.
- If the bounds of a base NDF are to be altered and retention of the pixel values of any of its components is not required, then a call to NDF_RESET should be made before calling this routine. This will eliminate any unnecessary processing which might be needed to retain the existing values. This step is not necessary with an NDF section, as no processing of pixel values takes place in this case.

NDF_SCOPY
Selectively copy NDF components to a new location

Description:

The routine propagates (copies) selected components of an NDF to a new location and returns an identifier for the resulting new base NDF.

Invocation:

```
CALL NDF_SCOPY( INDF1, CLIST, PLACE, INDF2, STATUS )
```

Arguments:

INDF1 = INTEGER (Given)
Identifier for the NDF (or NDF section) to be copied.

CLIST = CHARACTER * (*) (Given)
A comma-separated list of the NDF components which are to be propagated to the new data structure. By default, the HISTORY, LABEL and TITLE components are propagated. All extensions are also propagated by default except for any that have had a zero value assigned to the corresponding "PXT..." tuning parameter using NDF_TUNE. See the "Component Propagation" section for further details.

PLACE = INTEGER (Given and Returned)
An NDF placeholder (e.g. generated by the NDF_PLACE routine) which indicates the position in the data system where the new NDF will reside. The placeholder is annulled by this routine, and a value of NDF__NOPL will be returned (as defined in the include file NDF_PAR).

INDF2 = INTEGER (Returned)
Identifier for the new NDF.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If this routine is called with STATUS set, then a value of NDF__NOID will be returned for the INDF2 argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. In either event the placeholder will still be annulled. The NDF__NOID constant is defined in the include file NDF_PAR.

Component Propagation :

- The components whose values are to be propagated (copied) from the existing NDF to the new data structure are specified via the CLIST argument. Thus CLIST='DATA,QUALITY' would cause the DATA and QUALITY components to be propagated (if available) from the existing NDF to the new structure, in addition to those propagated by default. Component propagation may be suppressed by supplying a component name with the prefix 'NO'. Thus CLIST='DATA,NOHISTORY' would propagate the DATA component, but suppress propagation of HISTORY. If component names appear more than once in the CLIST value, then the last occurrence takes precedence.

- Propagation of specific NDF extensions may be suppressed by using 'NOEXTENSION()' as one of the items in the CLIST argument; a list of the extensions to be suppressed should appear between the parentheses. Thus CLIST='AXIS,NOEXTENSION(IRAS,ASTERIX)' would propagate the AXIS component, but suppress propagation of the IRAS and ASTERIX extensions (if present). Propagation of suppressed extensions may be re-enabled by specifying 'EXTENSION()' in a similar manner at a later point in the CLIST value.
- An asterisk (∗) may be used as a wild card to match all extension names. Thus 'NOEXTENSION(∗),EXTENSION(may be used to indicate that only the IRAS extension should be propagated.
- Whether or not a named extension is propagated by default can be controlled via an NDF tuning parameter (see NDF_TUNE). The defaults established using NDF_TUNE can be over-ridden by specifying the extension explicitly within the CLIST parameter; e.g. 'EXTENSION(FITS)' or 'NOEXTENSION(FITS)' can be used to over-ride the default established by the PXTFITS tuning parameter.
- Component names in the CLIST argument may be abbreviated to 3 characters, but extension names must appear in full.

NDF_SCTYP
Obtain the numeric type of a scaled NDF array component

Description:

The routine returns the numeric type of a scaled NDF array component as an upper-case character string (e.g. '_REAL'). The returned type describes the values stored in the array, before they are unscaled using the associated scale and zero values. Use NDF_TYPE if you need the data type of the array after it has been unscaled.

Invocation:

```
CALL NDF_SCTYP( INDF, COMP, TYPE, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the NDF array component whose type is required: 'DATA' or 'VARIANCE'.

TYPE = CHARACTER * (*) (Returned)
Numeric type of the component.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If the array is not stored in SCALED form, then this routine returns the same type as the ARY_TYPE routine.
- A comma-separated list of component names may also be supplied to this routine. In this case the result returned will be the lowest precision numeric type to which all the specified components can be converted without unnecessary loss of information.
- The symbolic constant NDF__SZTYP may be used for declaring the length of a character variable which is to hold the numeric type of an NDF array component. This constant is defined in the include file NDF_PAR.

NDF_SECT
Create an NDF section

Description:

The routine creates a new NDF section which refers to a selected region of an existing NDF (or NDF section). The region may be larger or smaller in extent than the initial NDF.

Invocation:

```
CALL NDF_SECT( INDF1, NDIM, LBND, UBND, INDF2, STATUS )
```

Arguments:

INDF1 = INTEGER (Given)
Identifier for the initial NDF.

NDIM = INTEGER (Given)
Number of dimensions for the new section.

LBND(NDIM) = INTEGER (Given)
Lower pixel-index bounds of the section.

UBND(NDIM) = INTEGER (Given)
Upper pixel-index bounds of the section.

INDF2 = INTEGER (Returned)
Identifier for the new section.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- The number of section dimensions need not match the number of dimensions in the initial NDF. Pixel-index bounds will be padded with `1's` as necessary to identify the pixels to which the new section should refer.
- The array components of sections which extend beyond the pixel-index bounds of the initial NDF will be padded with bad pixels.
- If the section bounds extend beyond the bounds of the associated base NDF and any of the `NDF's` axis arrays have defined values, then these values will be extrapolated as necessary.
- If this routine is called with STATUS set, then a value of NDF__NOID will be returned for the INDF2 argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The NDF__NOID constant is defined in the include file NDF_PAR.

NDF_SHIFT
Apply pixel-index shifts to an NDF

Description:

The routine applies pixel-index shifts to an NDF. An integer shift is applied to each dimension so that its pixel-index bounds, and the indices of each pixel, change by the amount of shift applied to the corresponding dimension. The NDF's pixels retain their values and none are lost.

Invocation:

```
CALL NDF_SHIFT( NSHIFT, SHIFT, INDF, STATUS )
```

Arguments:

NSHIFT = INTEGER (Given)
Number of dimensions to which shifts are to be applied. This must not exceed the number of NDF dimensions. If fewer shifts are applied than there are NDF dimensions, then the extra dimensions will not be shifted.

SHIFT(NSHIFT) = INTEGER (Given)
The pixel-index shifts to be applied to each dimension.

INDF = INTEGER (Given)
NDF identifier.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- Pixel-index shifts applied to a base NDF will affect the appearance of that NDF as seen by all base-NDF identifiers associated with it. However, NDF sections derived from that base NDF will remain unchanged (as regards both pixel-indices and array values).
- Pixel-index shifts applied to an NDF section only affect that section itself, and have no effect on other NDF identifiers.
- Pixel-index shifts cannot be applied to a base NDF while any of its components (or any of its axis arrays) is mapped for access, even through another identifier.
- Pixel-index shifts cannot be applied to an NDF section while any of its components (or any of its axis arrays) is mapped for access through the identifier supplied to this routine.

NDF_SIZE
Determine the size of an NDF

Description:

The routine returns the number of pixels in the NDF whose identifier is supplied (i.e. the product of its dimensions).

Invocation:

```
CALL NDF_SIZE( INDF, NPIX, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

NPIX = INTEGER (Returned)
Number of pixels in the NDF.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If this routine is called with STATUS set, then a value of 1 will be returned for the NPIX argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason.

NDF_SQMF
Set a new logical value for an NDF's quality masking flag

Description:

The routine sets a new logical value for an NDF's quality masking flag. This flag determines whether the NDF's quality component (if present) will be used to generate "bad" pixel values for automatic insertion into the data and variance arrays when these are accessed in READ or UPDATE mode. If this flag is set to .TRUE., then masking will occur, so that an application need not consider the quality information explicitly. If the flag is set to .FALSE., then automatic masking will not occur, so that the application can process the quality component by accessing it directly.

Invocation:

```
CALL NDF_SQMF( QMF, INDF, STATUS )
```

Arguments:

QMF = LOGICAL (Given)
The logical value to be set for the quality masking flag.

INDF = INTEGER (Given)
NDF identifier.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- A quality masking flag is associated with each NDF identifier and is initially set to .TRUE.. Its value changes to .FALSE. whenever the quality component is accessed directly (e.g. using NDF_MAP or NDF_MAPQL) and reverts to .TRUE. when access is relinquished (e.g. using NDF_UNMAP). This default behaviour may also be over-ridden by calling NDF_SQMF to set its value explicitly. The routine NDF_QMF allows the current value to be determined.
- The value of the quality masking flag is not propagated to new identifiers.

NDF_SSARY
Create an array section, using an NDF section as a template

Description:

The routine creates a "similar section" from an array (whose ARY_ system identifier is supplied) using an existing NDF section as a template. An identifier for the array section is returned and this may subsequently be manipulated using the ARY_ system routines (SUN/11). The new array section will bear the same relationship to its base array as the NDF template does to its own base NDF. Allowance is made for any pixel-index shifts which may have been applied, so that the pixel-index system of the new array section matches that of the NDF template.

This routine is intended for use when an array which must match pixel-for-pixel with an NDF is stored in an NDF extension; if an NDF section is obtained, then this routine may be used to obtain a pixel-by-pixel matching section from the array.

Invocation:

```
CALL NDF_SSARY( IARY1, INDF, IARY2, STATUS )
```

Arguments:

IARY1 = INTEGER (Given)
The ARY_ system identifier for the array, or array section, from which the new section is to be drawn.

INDF = INTEGER (Given)
NDF_ system identifier for the template NDF section (may also be a base NDF).

IARY2 = INTEGER (Returned)
ARY_ system identifier for the new array section.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- This routine will normally generate an array section. However, if the input array is a base array and the input NDF is a base NDF with the same pixel-index bounds, then there is no need to generate a section in order to access the required part of the array. In this case, a base array identifier will be returned instead.
- It is the caller's responsibility to annul the ARY_ system identifier issued by this routine (e.g. by calling ARY_ANNUL) when it is no longer required. The NDF_ system will not perform this task itself.
- The new array generated by this routine will have the same number of dimensions as the array from which it is derived. If the template NDF has fewer dimensions, then the pixel-index bounds of the extra array dimensions are preserved unchanged. If the NDF has more dimensions, then the extra ones are ignored.
- This routine takes account of the transfer windows of the array and NDF supplied and will restrict the transfer window of the new array section so as not to grant access to regions of the base array which were not previously accessible through both the input array and the NDF section.
- If this routine is called with STATUS set, then a value of ARY__NOID will be returned for the IARY2 argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The ARY__NOID constant is defined in the include file ARY_PAR.

NDF_STATE
Determine the state of an NDF component (defined or undefined)

Description:

The routine returns a logical value indicating whether an NDF component has a defined value (or values).

Invocation:

```
CALL NDF_STATE( INDF, COMP, STATE, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the component; any NDF component name is valid.

STATE = LOGICAL (Returned)
Whether the specified component is defined.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If a component name of 'EXTENSION' is given, then a .TRUE. result will be returned if one or more extensions are present in the NDF.
- A comma-separated list of component names may also be given, in which case the routine will return the logical "AND" of the states of the specified components (i.e. a .TRUE. result will be returned only if all the components have defined values).

NDF_STYPE
Set a new type for an NDF array component

Description:

The routine sets a new full type for an NDF array component, causing its storage type to be changed. If the component's values are defined, they will be converted from from the old type to the new one. If they are undefined, then no conversion will be necessary. Subsequent enquiries will reflect the new type. Conversion may be performed between any types supported by the NDF_ routines, including from a non-complex type to a complex type (and vice versa).

Invocation:

```
CALL NDF_STYPE( FTYPE, INDF, COMP, STATUS )
```

Arguments:

FTYPE = CHARACTER * (*) (Given)
The new full type specification for the NDF component (e.g. '_REAL' or 'COMPLEX_INTEGER').

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the array component whose type is to be set: 'DATA' or 'VARIANCE'.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- The routine may only be used to change the type of a component of a base NDF. If it is called for an NDF which is not a base NDF, then it will return without action. No error will result.
- A comma-separated list of component names may also be supplied, in which case the type of each component will be set to the same value in turn.
- An error will result if a component being modified, or any part of it, is currently mapped for access (e.g. through another identifier).
- If the type of a component is to be changed without its values being retained, then a call to NDF_RESET should be made beforehand. This will avoid the cost of converting all the values.

NDF_TEMP
Obtain a placeholder for a temporary NDF

Description:

The routine returns an NDF placeholder which may be used to create a temporary NDF (i.e. one which will be deleted automatically once the last identifier associated with it is annulled). The placeholder returned by this routine may be passed to other routines (e.g. NDF_NEW or NDF_COPY) to produce a temporary NDF in the same way as a new permanent NDF would be created.

Invocation:

```
CALL NDF_TEMP( PLACE, STATUS )
```

Arguments:

PLACE = INTEGER (Returned)
Placeholder for a temporary NDF.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- Placeholders are intended only for local use within an application and only a limited number of them are available simultaneously. They are always annulled as soon as they are passed to another routine to create a new NDF, where they are effectively exchanged for an NDF identifier.
- If this routine is called with STATUS set, then a value of NDF__NOPL will be returned for the PLACE argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. The NDF__NOPL constant is defined in the include file NDF_PAR.

NDF_TUNE
Set an NDF_ system tuning parameter

Description:

The routine sets a new value for an NDF_ system internal tuning parameter.

Invocation:

```
CALL NDF_TUNE( VALUE, TPAR, STATUS )
```

Arguments:

VALUE = INTEGER (Given)
New value for the tuning parameter.

TPAR = CHARACTER * (*) (Given)
Name of the parameter to be set (case insensitive). This name may be abbreviated, to no less than 3 characters.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

The following tuning parameters are currently available:

- 'AUTO_HISTORY'; Controls whether to include an empty History component in NDFs created using NDF_NEW or NDF_CREAT. If the tuning parameter is zet to zero (the default), no History component will be included in the new NDFs. If the tuning parameter is zet non-zero, a History component will be added automatically to the new NDFs.
- 'DOCVT': Controls whether to convert foreign format data files to and from native NDF format for access (using the facilities described in SSN/20). If DOCVT is set to 1 (the default), and the other necessary steps described in SSN/20 have been taken, then such conversions will be performed whenever they are necessary to gain access to data stored in a foreign format. If DOCVT is set to 0, no such conversions will be attempted and all data will be accessed in native NDF format only. The value of DOCVT may be changed at any time. It is the value current when a dataset is first accessed by the NDF_ library which is significant.
- 'KEEP': Controls whether to retain a native format NDF copy of any foreign format data files which are accessed by the NDF_ library (and automatically converted using the facilities described in SSN/20). If KEEP is set to 0 (the default), then the results of converting foreign format data files will be stored in scratch filespace and deleted when no longer required. If KEEP is set to 1, the results of the conversion will instead be stored in permanent NDF data files in the default directory (such files will have the same name as the foreign file from which they are derived and a file type of '.sdf'). Setting KEEP to 1 may be useful if the same datasets are to be re-used, as it avoids having to convert them on each occasion. The value of KEEP may be changed at any time. It is the value current when a foreign format file is first accessed by the NDF_ library which is significant.
- 'ROUND': Specifies the way in which floating-point values should be converted to integer during automatic type conversion. If ROUND is set to 1, then floating-point values are rounded to the nearest integer value. If ROUND is set to 0 (the default), floating-point values are truncated towards zero.
- 'SHCVT': Controls whether diagnostic information is displayed to show the actions being taken to convert to and from foreign data formats (using the facilities described in SSN/20). If SHCVT is set to 1, then this information is displayed to assist in debugging external format

conversion software whenever a foreign format file is accessed. If SHCVT is set to 0 (the default), this information does not appear and format conversion proceeds silently unless an error occurs.

- 'TRACE': Controls the reporting of additional error messages which may occasionally be useful for diagnosing internal problems within the NDF_ library. If TRACE is set to 1, then any error occurring within the NDF_ system will be accompanied by error messages indicating which internal routines have exited prematurely as a result. If TRACE is set to 0 (the default), this internal diagnostic information will not appear and only standard error messages will be produced.
- 'WARN': Controls the issuing of warning messages when certain non-fatal errors in the structure of NDF data objects are detected. If WARN is set to 1 (the default), then a warning message is issued. If WARN is set to 0, then no message is issued. In both cases normal execution continues and no STATUS value is set.
- 'PXT...': Controls whether a named NDF extension should be propagated by default when NDF_PROP or NDF_SCOPY is called. The name of the extension should be appended to the string "PXT" to form the complete tuning parameter name. Thus the tuning parameter PXTFITS would control whether the FITS extension is propagated by default. If the value for the parameter is non-zero, then the extension will be propagated by default. If the value for the parameter is zero, then the extension will not be propagated by default. The default established by this tuning parameter can be over-ridden by specifying the extension explicitly within the CLIST argument when calling NDF_PROP or NDF_SCOPY. The default value for all "PXT..." tuning parameters is 1, meaning that all extensions are propagated by default.

NDF_TYPE
Obtain the numeric type of an NDF array component

Description:

The routine returns the numeric type of one of the array components of an NDF as an upper-case character string (e.g. '_REAL').

Invocation:

```
CALL NDF_TYPE( INDF, COMP, TYPE, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the NDF array component whose type is required: 'DATA', 'QUALITY' or 'VARIANCE'.

TYPE = CHARACTER * (*) (Returned)
Numeric type of the component.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- A comma-separated list of component names may also be supplied to this routine. In this case the result returned will be the lowest precision numeric type to which all the specified components can be converted without unnecessary loss of information.
- The numeric type of a scaled array is determined by the numeric type of the scale and zero terms, not by the numeric type of the underlying array elements.
- The value returned for the QUALITY component is always '_UBYTE'.
- The symbolic constant NDF__SZTYP may be used for declaring the length of a character variable which is to hold the numeric type of an NDF array component. This constant is defined in the include file NDF_PAR.

NDF_UNMAP
Unmap an NDF or a mapped NDF array

Description:

The routine unmaps an NDF, or a individual NDF array which has previously been mapped for READ, UPDATE or WRITE access.

Invocation:

```
CALL NDF_UNMAP( INDF, COMP, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

COMP = CHARACTER * (*) (Given)
Name of the NDF component to be unmapped: 'AXIS', 'DATA', 'QUALITY', 'VARIANCE' or '*'. The last value acts as a wild card, causing all mapped arrays to be unmapped.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- This routine attempts to execute even if STATUS is set on entry, although no further error report will be made if it subsequently fails under these circumstances.
- A component name of 'AXIS' will act as a partial wild card, unmapping any axis arrays which are mapped, but leaving other components unchanged. The routine NDF_AUNMP may be used to unmap individual axis arrays.
- A comma-separated list of component names may also be given, in which case each component will be unmapped in turn.
- An error will be reported if a component has not previously been mapped for access, except in the case where a value of '*' is given for COMP, or where 'AXIS' is used to unmap axis arrays.

NDF_VALID
Determine whether an NDF identifier is valid

Description:

The routine determines whether an NDF identifier is valid (i.e. associated with an NDF).

Invocation:

```
CALL NDF_VALID( INDF, VALID, STATUS )
```

Arguments:

INDF = INTEGER (Given)
Identifier to be tested.

VALID = LOGICAL (Returned)
Whether the identifier is valid.

STATUS = INTEGER (Given and Returned)
The global status.

NDF_XDEL
Delete a specified NDF extension

Description:

The routine deletes a named extension in an NDF together with its contents, if any. No error results if the specified extension does not exist.

Invocation:

```
CALL NDF_XDEL( INDF, XNAME, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

XNAME = CHARACTER * (*) (Given)
Name of the extension to be deleted.

STATUS = INTEGER (Given and Returned)
The global status.

NDF_XIARY
Obtain access to an array stored in an NDF extension

Description:

The routine locates an array stored in an NDF extension and imports it into the ARY_ system, returning an array identifier for it. If necessary, a section of the array will be selected so that it matches pixel-for-pixel with the main data array of the NDF (or NDF section) supplied. The returned array identifier may be used to manipulate the array using the ARY_ routines (see SUN/11).

Invocation:

```
CALL NDF_XIARY( INDF, XNAME, CMPT, MODE, IARY, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

XNAME = CHARACTER * (*) (Given)
Name of the extension.

CMPT = CHARACTER * (*) (Given)
Name of the array component within the extension.

MODE = CHARACTER * (*) (Given)
Mode of access required: 'READ', 'UPDATE' or 'WRITE'.

IARY = INTEGER (Returned)
Array identifier.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- The value given for the CMPT argument may be an HDS path name, consisting of several fields separated by '.', so that an object can be accessed in a sub-component (or a sub-sub-component...) of an NDF extension. Array subscripts may also be included. Thus a string such as 'FILTER(3).FLATFIELD' could be used as a valid CMPT value.
- This routine will normally generate an array section. However, if the input NDF is a base NDF and the requested array has the same pixel-index bounds, then there is no need to generate a section in order to access the required part of the array. In this case, a base array identifier will be issued instead.
- It is the caller's responsibility to annul the ARY_ system identifier returned by this routine (e.g. by calling ARY_ANNUL) when it is no longer required. The NDF_ system will not perform this task itself.
- The array associated with the returned identifier will have the same number of dimensions as the base array from which it is derived. If the input NDF has fewer dimensions than this, then the pixel-index bounds of the extra array dimensions are preserved unchanged. If the NDF has more dimensions, then the extra ones are ignored.
- This routine takes account of the transfer window of the NDF supplied and will restrict the transfer window of the new array section so as not to grant access to regions of the base array which are not accessible in the input NDF.
- If this routine is called with STATUS set, then a value of ARY__NOID will be returned for the IARY argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason.
- The ARY__NOID constant is defined in the include file ARY_PAR.

NDF_XGT0x
Read a scalar value from a component within a named NDF extension

Description:

The routine reads a scalar value from a component within a named NDF extension. The extension must already exist, although the component within the extension need not exist (a default value, established beforehand, will be returned if necessary).

Invocation:

```
CALL NDF_XGT0x( INDF, XNAME, CMPT, VALUE, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

XNAME = CHARACTER * (*) (Given)
Name of the NDF extension.

CMPT = CHARACTER * (*) (Given)
Name of the component within the extension whose value is to be obtained.

VALUE = ? (Given and Returned)
The value obtained from the extension component. Its type is determined by the name of the routine called (see the Notes section).

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- There is a routine for reading values with each of the standard Fortran data types: integer, real, double precision, logical and character. Replace the (lower case) "x" in the routine name by I, R, D, L or C as appropriate.
- The value given for the CMPT argument may be an HDS path name, consisting of several fields separated by '.', so that an object can be accessed in a sub-component (or a sub-sub-component...) of an NDF extension. Array subscripts may also be included. Thus a string such as 'CALIB.FILTER(3).WAVELENGTH' could be used as a valid CMPT value.
- If the requested component in the extension does not exist, then the VALUE argument will be returned unchanged. A suitable default should therefore be established before this routine is called.
- If the length of the character VALUE argument passed to the NDF_XGT0C routine is too short to accommodate the returned result without losing significant (non-blank) trailing characters, then this will be indicated by an appended ellipsis, i.e. '...'. No error will result.

NDF_XLOC
Obtain access to a named NDF extension via an HDS locator

Description:

The routine returns an HDS locator to a named extension (if present) in an NDF. An error results if the specified extension is not present.

Invocation:

```
CALL NDF_XLOC( INDF, XNAME, MODE, LOC, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

XNAME = CHARACTER * (*) (Given)
Name of the required extension.

MODE = CHARACTER * (*) (Given)
Mode of access required to the extension: 'READ', 'UPDATE' or 'WRITE'.

LOC = CHARACTER * (*) (Returned)
Extension locator.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If WRITE access is specified, then any existing extension contents or values will be erased or reset, so that the extension is ready to receive new values. If UPDATE access is specified, then existing values will be retained so that they may be modified.
- It is the caller's responsibility to annul the HDS locator issued by this routine (e.g. by calling DAT_ANNUL) when it is no longer required. The NDF_ system will not perform this task itself.
- Although this routine will check the access mode value supplied against the available access to the NDF, HDS does not allow the returned locator to be protected against write access in the case where WRITE access to the NDF is available, but only READ access was requested. In this case it is the responsibility of the caller to respect the locator access restriction.
- If this routine is called with STATUS set, then an invalid locator will be returned for the LOC argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason.

NDF_XNAME
Obtain the name of the N'th extension in an NDF

Description:

The routine returns the name of the N'th extension in an NDF. If the requested extension does not exist, then the name is returned blank. The routine may therefore be used to obtain the names of all the extensions present by setting N to 1,2... etc. until a blank name is returned. Note that the order in which these names are obtained is not defined.

Invocation:

```
CALL NDF_XNAME( INDF, N, XNAME, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

N = INTEGER (Given)
The number of the extension whose name is required.

XNAME = CHARACTER * (*) (Returned)
The extension name (in upper case).

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- The symbolic constant NDF__SZXNM is provided to define the length of character variables which are to hold an NDF extension name. This constant is defined in the include file NDF_PAR.

NDF_XNEW
Create a new extension in an NDF

Description:

The routine creates a new named extension of specified type and shape in an NDF structure, and returns an HDS locator to it.

Invocation:

```
CALL NDF_XNEW( INDF, XNAME, TYPE, NDIM, DIM, LOC, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

XNAME = CHARACTER * (*) (Given)
Extension name.

TYPE = CHARACTER * (*) (Given)
HDS data type of the extension.

NDIM = INTEGER (Given)
Number of extension dimensions.

DIM(NDIM) = INTEGER (Given)
Extension dimension sizes.

LOC = CHARACTER * (*) (Returned)
Locator to the newly created extension.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If this routine is called with STATUS set, then an invalid locator will be returned for the LOC argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason.

NDF_XNUMB
Determine the number of extensions in an NDF

Description:

The routine returns the number of extensions present in the NDF whose identifier is supplied.

Invocation:

```
CALL NDF_XNUMB( INDF, NEXTN, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

NEXTN = INTEGER (Returned)
Number of extensions present.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- If this routine is called with STATUS set, then a value of zero will be returned for the NEXTN argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason.

NDF_XPT0x
Write a scalar value to a component within a named NDF extension

Description:

The routine writes a scalar value to a component within a named NDF extension. The extension must already exist, although the component within the extension need not exist and will be created if necessary.

Invocation:

```
CALL NDF_XPT0x( VALUE, INDF, XNAME, CMPT, STATUS )
```

Arguments:

VALUE = ? (Given)
The value to be written to the extension component. Its type is determined by the name of the routine called (see the Notes section).

INDF = INTEGER (Given)
NDF identifier.

XNAME = CHARACTER * (*) (Given)
Name of the NDF extension.

CMPT = CHARACTER * (*) (Given)
Name of the component within the extension whose value is to be assigned.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- There is a routine for writing values with each of the standard Fortran data types: integer, real, double precision, logical and character. Replace the (lower case) "x" in the routine name by I, R, D, L or C as appropriate.
- The value given for the CMPT argument may be an HDS path name, consisting of several fields separated by '.', so that an object can be accessed in a sub-component (or a sub-sub-component...) of an NDF extension. Array subscripts may also be included. Thus a string such as 'CALIB.FILTER(3).WAVELENGTH' could be used as a valid CMPT value.
- All HDS structures which lie above the specified component within the extension must already exist, otherwise an error will result.
- If the specified extension component does not already exist, then it will be created by this routine. If it already exits, but does not have the correct type or shape, then it will be deleted and a new scalar component with the appropriate type will be created in its place.

NDF_XSTAT
Determine if a named NDF extension exists

Description:

The routine returns a logical value indicating whether a named extension is present in an NDF.

Invocation:

```
CALL NDF_XSTAT( INDF, XNAME, THERE, STATUS )
```

Arguments:

INDF = INTEGER (Given)
NDF identifier.

XNAME = CHARACTER * (*) (Given)
Name of the extension.

THERE = LOGICAL (Returned)
Whether the extension is present in the NDF.

STATUS = INTEGER (Given and Returned)
The global status.

NDF_ZDELT
Create a compressed copy of an NDF using DELTA compression

Description:

The routine creates a copy of the supplied NDF, and then compresses selected array components within the copy using delta compression. This compression is lossless, but only operates on arrays holding integer values. It assumes that adjacent integer values in each input array tend to be close in value, and so differences between adjacent values can be represented in fewer bits than the absolute values themselves. The differences are taken along a nominated pixel axis within the supplied array (specified by argument ZAXIS). Any input value that differs from its earlier neighbour by more than the data range of the selected data type is stored explicitly using the data type of the input array.

Further compression is achieved by replacing runs of equal input values by a single occurrence of the value with a correspsonding repetition count.

It should be noted that the degree of compression achieved is dependent on the nature of the data, and it is possible for a compressed array to occupy more space than the uncompressed array. The mean compression factor actually achieved is returned in argument ZRATIO (the ratio of the supplied NDF size to the compressed NDF size).

Invocation:

```
CALL NDF_ZDELT( INDF1, COMP, MINRAT, ZAXIS, TYPE, PLACE, INDF2, ZRATIO, STATUS )
```

Arguments:

INDF1 = INTEGER (Given)
Identifier for the input NDF.

COMP = CHARACTER * (*) (Given)
A comma-separated list of component names to be compressed. These may be selected from 'DATA, 'VARIANCE' or 'QUALITY'. Additionally, if '*' is supplied all three components will be compressed.

MINRAT = REAL (Given)
The minimum allowed compression ratio for an array (the ratio of the supplied array size to the compressed array size). If compressing an array results in a compression ratio smaller than or equal to MINRAT, then the array is left uncompressed in the returned NDF. If the supplied value is zero or negative, then each array will be compressed regardless of the compression ratio.

ZAXIS = INTEGER (Given)
The index of the pixel axis along which differences are to be taken. If this is zero, a default value will be selected for each array that gives the greatest compression. An error will be reported if a value less than zero or greater than the number of axes in the input array is supplied.

TYPE = CHARACTER * (*) (Given)
The data type in which to store the differences between adjacent array values. This must be one of '_BYTE', '_WORD' or '_INTEGER'. Additionally, a blank string may be supplied in which case a default value will be selected for each array that gives the greatest compression.

PLACE = INTEGER (Given and Returned)
An NDF placeholder (e.g. generated by the NDF_PLACE routine) which indicates the position in the data system where the new NDF will reside. The placeholder is annulled by this routine, and a value of NDF__NOPL will be returned (as defined in the include file NDF_PAR).

INDF2 = INTEGER (Returned)
Identifier for the new NDF.

ZRATIO = _REAL (Returned)
The mean compression ratio actually achieved for the array components specified by COMP. The compression ratio for an array is the ratio of the number of bytes needed to hold the numerical values in the supplied array, to the number of bytes needed to hold the numerical values in the compressed array. It does not include the small overheads associated with the extra labels, etc, needed to store compressed data, and so may be inaccurate for small arrays. Additionally, this ratio only takes the specified array components into account. If the NDF contains significant quantities of data in other components, then the overall compression of the whole NDF will be less.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- An error is reported if any of the arrays to be compressed holds floating point values. The exception is that floating point values that are stored as scaled integers (see NDF_ZSCAL) are accepted.
- If the input NDF is already stored in DELTA format, it will be uncompressed and then recompressed using the supplied parameter values.
- An error will result if any of the specified array components of the input NDF are currently mapped for access.
- Complex arrays cannot be compressed using this routine. An error will be reported if the input NDF has a complex type, or if "TYPE" represents a complex data type.
- Delta compressed arrays are read-only. An error will be reported if an attempt is made to map a delta compressed array for WRITE or UPDATE access.
- When delta compressed arrays are mapped for READ access, uncompression occurs automatically. The pointer returned by NDF_MAP provides access to the uncompressed array values.
- The result of copying a compressed NDF (for instance, using NDF_PROP, etc.) will be an equivalent uncompressed NDF.
- When applied to a compressed NDF, the NDF_TYPE and NDF_FTYPE routines return information about the data type of the uncompressed NDF.
- If this routine is called with STATUS set, then a value of NDF__NOID will be returned for the INDF2 argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. In either event, the placeholder will still be annulled. The NDF__NOID constant is defined in the include file NDF_PAR.

NDF_ZSCAL
Create a compressed copy of an NDF using SCALE compression

Description:

The routine creates a new NDF holding a compressed copy of the supplied NDF. The compression is performed by scaling the DATA and VARIANCE arrays using a simple linear scaling, and then casting the scaled values into the specified data type. The amount of compression, and the amount of information lost, is thus determined by the input and output data types. For instance, if the input NDF is of type _DOUBLE (eight bytes) and the output is of type _WORD (two bytes), the compression ratio for each array component will be four to one.

Invocation:

```
CALL NDF_ZSCAL( INDF1, TYPE, SCALE, ZERO, PLACE, INDF2, STATUS )
```

Arguments:

INDF1 = INTEGER (Given)
Identifier for the input NDF.

TYPE = CHARACTER * (*) (Given)
Numeric type of the output NDF's DATA component (e.g. '_REAL' or '_INTEGER').

SCALE(2) = DOUBLE PRECISION (Given and Returned)
The scale factors to use when compressing the array components in the supplied NDF. The DATA array will be scaled using SCALE(1) and the VARIANCE array - if present - will be scaled using SCALE(2). If either of these is set to VAL__BADD, then a suitable value will be found automatically by inspecting the supplied array values. The values actually used will be returned on exit. See "Notes:" below. On exit, any supplied values will be rounded to values that can be represented accurately in the data type of the input NDF.

ZERO(2) = DOUBLE PRECISION (Given and Returned)
The zero offsets to use when compressing the array components in the supplied NDF. The DATA array will be offset using ZERO(1) and the VARIANCE array - if present - will be offset using ZERO(2). If either of these is set to VAL__BADD, then a suitable value will be found automatically by inspecting the supplied array values. The values actually used will be returned on exit. See "Notes:" below. On exit, any supplied values will be rounded to values that can be represented accurately in the data type of the input NDF.

PLACE = INTEGER (Given and Returned)
An NDF placeholder (e.g. generated by the NDF_PLACE routine) which indicates the position in the data system where the new NDF will reside. The placeholder is annulled by this routine, and a value of NDF__NOPL will be returned (as defined in the include file NDF_PAR).

INDF2 = INTEGER (Returned)
Identifier for the new NDF.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

- The compressed data may not be of type _DOUBLE. An error will be reported if TYPE is '_DOUBLE'.

- Only arrays that are stored in SIMPLE or PRIMITIVE form can be compressed. An error is reported if any other storage form is encountered whilst compressing the input NDF.
- The uncompressed array values are obtained by multiplying the compressed values by SCALE and then adding on ZERO.
- The default scale and zero values (used if VAL__BADD values are supplied for SCALE and/or ZERO) are chosen so that the extreme array values will fit into the dynamic range of the output data type, allowing a small safety margin.
- Complex arrays cannot be compressed using this routine. An error will be reported if the input NDF has a complex type, or if "TYPE" represents a complex data type.
- The resulting NDF will be read-only. An error will be reported if an attempt is made to map it for WRITE or UPDATE access.
- When the output NDF is mapped for READ access, uncompression occurs automatically. The pointer returned by NDF_MAP provides access to the uncompressed array values.
- The result of copying a compressed NDF (for instance, using NDF_PROP, etc.) will be an equivalent uncompressed NDF. SCALE(2) = DOUBLE PRECISION (Given)
- When applied to a compressed NDF, the NDF_TYPE and NDF_FTYPE routines return information about the data type of the uncompressed NDF.
- If this routine is called with STATUS set, then a value of NDF__NOID will be returned for the INDF2 argument, although no further processing will occur. The same value will also be returned if the routine should fail for any reason. In either event, the placeholder will still be annulled. The NDF__NOID constant is defined in the include file NDF_PAR.

E THE NDF_ LIBRARY C INTERFACE

E.1 Conventions Used in the C Interface

As of Version 2.0, the NDF_ library is implemented in C, with an additional layer providing the traditional Fortran interface described in the rest of this manual. This appendix describes the underlying C interface, which allows NDF functions to be called from programs written in C (see Appendix F for a list of the C functions available). It is quite easy to translate the Fortran descriptions and examples given in the rest of this manual into C once you are aware of the conventions used. The following notes are intended to assist with this:

(1) C function names are derived from the corresponding Fortran routine names by removing the underscore, converting to lower case and then capitalising the fourth character. Thus, the Fortran routine NDF_ABCDEF becomes the C function ndfAbcdef.

(2) A single header file "ndf.h" is provided to define the C interface. This contains function prototypes together with C equivalent definitions for all the symbolic constants and error codes used by the Fortran version. All the constants use exactly the same names (in upper case) as in Fortran. It is recommended that you always include this file, since some functions may be implemented via C macros and will therefore only be available in this way.

(3) The data types used in Fortran and C for routine arguments and returned values are related as follows:

Fortran Type	C Type
DOUBLE PRECISION	double
REAL	float
INTEGER	int
LOGICAL	int
CHARACTER	char *

Note that the C interface uses "int" for both the Fortran INTEGER and LOGICAL data types, but interprets the latter according to whether the C integer is non-zero or zero (corresponding with Fortran's .TRUE. and .FALSE. respectively).

(4) Input scalar arguments are always passed in C by value.

(5) Non-scalar input arguments (typically strings) are passed in C by pointer, qualified by "const" as appropriate.

(6) All output arguments are passed in C by pointer. In the case of output array arguments, the caller must ensure that adequate space has been allocated to receive the returned values (also see the notes on passing character strings below). The C array notation "[ ]" is used in the function prototypes to indicate where a returned value is an array rather than a scalar value.

(7) All C strings should be null-terminated.

(8) Whenever a C string value is to be returned (via a argument "arg", say, with type char ∗), the argument is followed in C by an associated integer length argument ("arg_length") which does not appear in the Fortran version of the routine. This additional argument specifies the length of the character array allocated to receive the string, including the final null. Truncation of the returned string will occur if this length would be exceeded.

(9) An array of character strings is passed in C as an array of pointers to null-terminated strings (like the standard argument vector passed to the C "main" function).

(10) A few functions pass HDS locators. These are stored in variables of type `HDSLoc *`. The calling function should include the `hds.h` header file.

(11) Strings which are used to describe the data type of NDF components must contain the same text in C as in Fortran. Hence, you should continue to use "_REAL", for example, (and not "_FLOAT") when specifying the data type of a new NDF.

(12) When mapping NDF array components, the C interface will usually return a pointer to void, reflecting the fact that the data type is determined at run time and is therefore not known to the mapping function. To access the mapped data, you should cast this pointer to the appropriate pointer type before use, as follows:

Mapped Data Type	C Pointer Cast
_DOUBLE	(double ∗)
_REAL	(float ∗)
_INTEGER	(int ∗)
_WORD	(short ∗)
_UWORD	(unsigned short ∗)
_BYTE	(signed char ∗)
_UBYTE	(unsigned char ∗)

(13) Remember that the data storage order used when mapping multi-dimensional array data follows the Fortran convention (where the first array index varies most rapidly) and not the C convention (where the final array index varies most rapidly).

(14) Several functions pass pointers to Objects defined by the AST library (SUN/211) for handling world coordinate system information. These use the same C argument passing conventions for these pointers as used in the AST library itself.

E.2 Multi-threaded Applications

In a multi-threaded context, each base NDF is either "locked" by a specific thread, or "unlocked". Locking is a long term contract that persists between invocations of public NDF functions, and can be changed only by the caller using one of the public lock-management methods: ndfLock, ndfUnlock and ndfLocked.

A locking contract means that:

- only the thread with the lock can invokes public NDF functions that read, write or modify the base NDF through any identifier.
- if a thread attempts to invoke a public NDF function to read, write or modify an NDF which it has not locked, an error is reported. The only exception is that ndfLock can be called on an NDF that is not locked by any thread (i.e. the identifier passed to ndfLock has been unlocked previously using ndfUnlock).

Upon creation, an NDF is always locked by the current thread. The locker thread can at any time unlock the NDF by calling ndfUnlock on any identifier for the NDF[30], allowing another thread subsequently to lock the NDF by calling ndfLock on the same identifier. An error is reported if the NDF is currently locked by a different thread. Calling ndfUnlock will unlock both the supplied identifier and the associated base NDF - an error is reported if the NDF is currently locked by another thread.

A consequence of this is that an NDF identifier will become unsable if another identifier for the same base NDF is unlocked. However, the identifier will become usable again if the original thread regains the lock on the NDF.

When an NDF identifier is unlocked, its associated context level is set to a special value indicating it is not in any context. All other identifiers for the same base NDF are left unchanged. When an NDF identifier is locked, its associated context level is set to the current context level in the thread that locks it. The ndfEnd function only annuls NDF identifiers for base NDFs that are locked by the current thread. The function ndfReport will report the lock state of all currently active NDF identifiers.

E.3 C-only Functions

This section contains descriptions of functions that are only available in the C interface.

[30]All other identifiers referring to the same base NDF remain valid but cannot be used without error until the original locker thread has regained the lock.

ndfLock
Lock an NDF for use by the current thread

Description:

This function locks the supplied NDF (both the supplied identifier and the associated base NDF) for sole use by the current thread. An error will be reported if the NDF is currently locked by another thread. Consequently, the NDF should be unlocked using ndfUnlock before calling this function.

Invocation:

```
void ndfLock( int indf, int *status )
```

Notes:

- This function returns without action if the NDF is already locked by the currently running thread.
- The supplied NDF identifier will be imported into the current NDF context within the currently running thread.
- This function attempts to execute even if " status" is set on entry, although no further error report will be made if it subsequently fails under these circumstances.

Parameters

indf

NDF identifier.

***status**

The global status.

ndfLocked
Return information about any lock on an NDF

Description:

This functions returns information about any lock on the base NDF associated with the supplied NDF identifier. NDFs are locked and unlocked using functions ndfLock and ndfUnlock.

Invocation:

```
result = ndfLocked( int indf, int *status )
```

Arguments:

indf

NDF identifier.

***status**

The global status.

Returned Value:

0 - the base NDF is unlocked

1 - the base NDF is locked by the current thread

-1 - the base NDF is locked by some other thread

Notes:

- This function attempts to execute even if " status" is set on entry, although no further error report will be made if it subsequently fails under these circumstances.

ndfReport
Report any currently active NDF identifiers

Description:

This function displays information about any currently active NDF identifiers.

Invocation:

```
result = ndfReport( int report, int *status );
```

Arguments:

report

If non-zero, an informational message listing any currently active NDF identifiers is displayed. Nothing is displayed if there are no active identifiers, or if report is zero.

***status**

The global status.

Returned Value:

The number of active NDF identifiers found.

Notes:

- This function attempts to execute even if " status" is set on entry, although no further error report will be made if it subsequently fails under these circumstances.

ndfUnlock
Unlock an NDF so it can then be locked by another thread

Description:

This function unlocks the supplied NDF (both the supplied identifier and the associated base NDF) so that another thread can then lock it for its own use using function ndfLock. After calling this function, an error will be reported if an attempt is made to access the NDF in any way, either through the supplied identifier or any other identifier that refers to the same base NDF. There are however two exceptions to this rule:

- An unlocked NDF identifier can be locked using ndfLock, allowing the thread full access to the NDF using the locked identifier.
- An unlocked NDF identifier can be annulled using ndfAnnul.

Invocation:

```
void ndfUnlock( int indf, int *status )
```

Notes:

- This function returns without action if the NDF is already unlocked.
- This function will report an error if the supplied NDF is locked by any thread other than the currently running thread.
- The supplied NDF identifier will be removed from the NDF context associated with the currently running thread, and placed in a " null" context that is ignored by the ndfEnd function. The ndfReport function can be used to determine if any NDF identifiers are currently in this " null" context.
- This function attempts to execute even if " status" is set on entry, although no further error report will be made if it subsequently fails under these circumstances.

Parameters

indf

NDF identifier.

***status**

The global status.

E.4 Building C Applications

Building C applications which call the NDF_ library differs very little from building Fortran applications and is covered in §24.

F ALPHABETICAL LIST OF C FUNCTIONS

This section gives a complete list of the functions available via the C interface to the NDF_ library, in the form of ANSI C function prototypes. For details of each function, see the Fortran routine description in Appendix D.

For general information about the C interface, see Appendix E.

void ndfAcget(int indf, const char *comp, int iaxis, char *value, int value_length, int *status);

Obtain the value of an NDF axis character component

void ndfAclen(int indf, const char *comp, int iaxis, int *length, int *status);
Determine the length of an NDF axis character component

void ndfAcmsg(const char *token, int indf, const char *comp, int iaxis, int *status);
Assign the value of an NDF axis character component to a message token

void ndfAcput(const char *value, int indf, const char *comp, int iaxis, int *status);
Assign a value to an NDF axis character component

void ndfAcre(int indf, int *status);
Ensure that an axis coordinate system exists for an NDF

void ndfAunmp(int indf, const char *comp, int iaxis, int *status);	*Unmap an NDF axis array component*

void ndfBad(int indf, const char *comp, int check, int *bad, int *status);	*Determine if an NDF array component may contain bad pixels*
void ndfBase(int indf1, int *indf2, int *status);	*Obtain an identifier for a base NDF*
void ndfBb(int indf, unsigned char *badbit, int *status);	*Obtain the bad-bits mask value for the quality component of an NDF*
void ndfBegin(void);	*Begin a new NDF context*
void ndfBlock(int indf1, int ndim, const int mxdim[ ], int iblock, int *indf2, int	*Obtain an NDF section containing a block of adjacent pixels*

void ndfCrepl(const char *param, int *place, int *status); *Create a new NDF placeholder via the ADAM parameter system*

void ndfDelet(int *indf, int *status); *Delete an NDF*

void ndfDim(int indf, int ndimx, int dim[ ], int *ndim, int *status); *Enquire the dimension sizes of an NDF*

void nd-fEnd(int *status); *End the current NDF context*

void nd-fEx-ist(const char *param, const char *mode, int *indf, int *status); *See if an existing NDF is associated with an ADAM parameter.*

void ndfFind(const char loc[DAT__SZLOC], const char *name, int *indf, int *status); *Find an NDF in an HDS structure and import it into the NDF_ system*

∗**status);**

Get compression details for a DELTA compressed NDF array component

void ndfGtszd(int indf, const char *comp, double *scale, double *zero, int *status); *Get scale and zero factors for a scaled array in an NDF*

void ndfGtszi(int indf, const char *comp, int *scale, int *zero, int *status); *Get scale and zero factors for a scaled array in an NDF*

void ndfGtszr(int indf, const char *comp, float *scale, float *zero, int *status); *Get scale and zero factors for a scaled array in an NDF*

void ndfGtune(const char *tpar, *Obtain the value of an NDF_ system tuning parameter*

void ndfHpurg(int indf, int irec1, int irec2, int *status); *Delete a range of records from an NDF history component*

void ndfHput(const char *hmode, const char *appn, int repl, int nlines, char *const text[ ], int trans, int wrap, int rjust, int indf, int *status); *Write history information to an NDF*

void ndfHsmod(const char *hmode, int indf, int *status); *Set the history update mode for an NDF*

void ndfHsdat(const char *datee, int indf, int *status); *Set the date and time for the next history record in an NDF*

void ndfIsacc(int indf, const char *access, int *isacc, *Determine whether a specified type of NDF access is available*

void ndfMtypn(const char ∗typlst, int n, const int ndfs[ ], const char ∗comp, char ∗itype, int itype_length, char ∗dtype, int dtype_length, int ∗status);

Match the types of the array components of a number of NDFs

void
ndfN-
bloc(
int
indf,
int
ndim,
const
int
mxdim[ ],
int
∗nblock,
int
∗status
);

Determine the number of blocks of adjacent pixels in an NDF

void
ndfNchnk(
int
indf,
int
mx-
pix,
int
∗nchunk,
int
∗status
);

Determine the number of chunks of contiguous pixels in an NDF

void
ndfNew(
const
char
∗ftype,
int
ndim,
const
int
lbnd[ ],
const
int
ubnd[ ],
int
∗place,
int
∗indf,
int
∗status
);

Create a new simple NDF

void
ndfNewp(
const
char
∗ftype,
int
ndim,
const
int
ubnd[ ],

Create a new primitive NDF

void ndf- Sect(int indf1, int ndim, const int lbnd[ ], const int ubnd[ ], int *indf2, int *status); *Create an NDF section*

void ndf-Shift(int nshift, const int shift[ ], int indf, int *status);	*Apply pixel-index shifts to an NDF*
void ndf-Size(int indf, int *npix, int *status);	*Determine the size of an NDF*
void ndf-Sqmf(int qmf, int indf, int *status);	*Set a new logical value for an NDF's quality masking flag*
void ndf-S-sary(int iary1, int indf, int *iary2, int *status);	*Create an array section, using an NDF section as a template*
void ndf-S-tate(int indf, const char *comp, int	*Determine the state of an NDF component (defined or undefined)*

void nd- fX- i- ary(int indf, const char ∗xname, const char ∗cmpt, const char ∗mode, int ∗iary, int ∗status);

Obtain access to an array stored in an NDF extension

void ndfXloc(int indf, const char *xname, const char *mode, char loc[DAT__SZLOC], int *status); *Obtain access to a named NDF extension via an HDS locator*

void ndfXname(int indf, int n, char *xname, int xname_length, int *status); *Obtain the name of the N'th extension in an NDF*

void ndfXnew(int indf, const char *xname, const char *type, int ndim, const int dim[ ], char loc[DAT__SZLOC], int *status); *Create a new extension in an NDF*

void

G OBSOLETE ROUTINES

The routines described below have been rendered obsolete by developments in the NDF_ library and should not be used in new software. They are included here simply as an aid to understanding existing software which uses them, and to allow them to be replaced with alternative techniques as the opportunity arises. These routines will eventually be removed from the NDF_ documentation and may, in some cases, eventually be eliminated from the library altogether.

The reason for obsolescence is indicated in each case.

NDF_IMPRT
Import an NDF into the NDF_ system from HDS

Description:

The routine imports an NDF into the NDF_ system from HDS and issues an identifier for it. The NDF may then be manipulated by the NDF_ routines.

Invocation:

```
CALL NDF_IMPRT( LOC, INDF, STATUS )
```

Arguments:

LOC = CHARACTER * (*) (Given)
HDS locator to an NDF structure.

INDF = INTEGER (Returned)
NDF identifier.

STATUS = INTEGER (Given and Returned)
The global status.

Notes:

This routine is obsolete. The same effect can be obtained by calling NDF_FIND with its second (NAME) argument set to a blank string.

NDF_TRACE
Set the internal NDF_ system error-tracing flag

Description:

The routine sets an internal flag in the NDF_ system which enables or disables error-tracing messages. If this flag is set to .TRUE., then any error occurring within the NDF_ system will be accompanied by error messages indicating which internal routines have exited prematurely as a result. If the flag is set to .FALSE., this internal diagnostic information will not appear and only standard error messages will be produced.

Invocation:

```
CALL NDF_TRACE( NEWFLG, OLDFLG )
```

Arguments:

NEWFLG = LOGICAL (Given)
The new value to be set for the error-tracing flag.

OLDFLG = LOGICAL (Returned)
The previous value of the flag.

Notes:

This routine is obsolete. The internal error tracing flag (referred to above) corresponds with the TRACE tuning parameter used by NDF_TUNE and NDF_GTUNE, so the same effect can be obtained by substituting these two routines.

H CHANGES AND NEW FEATURES

H.1 Changes Introduced in V1.3

The following describes the most significant changes which occurred in the NDF_ system between versions V1.2 and V1.3 (not the current version):

(1) New facilities have been added for handling NDF *history* information (see §22).

(2) New facilities have been added to allow the automatic reading and writing of data files written in a variety of "foreign" (*i.e.* non-NDF) formats. These are described in a separate document (SSN/20).

(3) A new routine NDF_OPEN has been added to provide a general means of accessing NDF datasets by name, locator, or a combination of both. It is modelled on the Fortran OPEN statement (see §20.10) and provides flexible NDF access for programmers who do not wish to use the ADAM parameter system.

(4) The symbolic constant DAT__ROOT provided by HDS is now supported by all NDF_ routines which accept HDS locators (see §20). Use of this constant in place of an HDS locator indicates that the associated component name is in fact the full name of the HDS object (or NDF). This allows access to HDS objects by name as an alternative to using locators. The name of a foreign format data file may also be supplied using this mechanism (SSN/20).

(5) All routines that accept the names of pre-existing NDF datasets now support subscripting, and will return an appropriate NDF section.

(6) A new selective copy routine NDF_SCOPY has been added (see §20.9) which performs component propagation in a similar manner to NDF_PROP but does not depend on the ADAM parameter system.

(7) The two sets of routines NDF_XGT0x and NDF_XPT0x (where "x" is I, R, D, L or C) will now accept compound component names when reading or writing the values of objects in NDF extensions. This allows direct access to values stored within nested structures or arrays in extensions (see §§11.4 & 11.5). The routine NDF_XIARY has also been similarly enhanced.

(8) The routine NDF_TUNE has been extended to support new tuning parameters, most of which are associated with the facilities for accessing foreign data formats (see above).

(9) Tuning parameters now acquire their default values from environment variables (see §23.3).

(10) Due to changes in the underlying data system (HDS), locators to data objects may now be annulled freely without risk of affecting the operation of the NDF_ library.

(11) There is no longer any need to call the routine HDS_START in standalone programs which use the NDF_ library (previously this was required), although doing so will do no harm.

(12) Instructions for compiling and linking NDF applications on UNIX have been added to the documentation.

(13) On UNIX systems where shareable libraries are supported, these are now installed in a separate .../share directory (rather than alongside the non-shareable libraries in the .../lib directory). You should include the appropriate .../share directory (normally /star/share on Starlink systems) in your library search path if you wish to use shareable libraries on UNIX.

(14) The routine NDF_XNUMB now returns a guaranteed value of zero if it is called with STATUS set, or if it should fail for any reason.

(15) A number of new error codes associated with the NDF *history* component and with access to foreign data formats have been added to the include file NDF_ERR.

(16) The routine NDF_IMPRT has been documented as obsolete. Its function is now performed by NDF_FIND by specifying a blank second argument.

(17) The routine NDF_TRACE has been documented as obsolete. Its function is now performed by NDF_TUNE via the tuning parameter 'TRACE'.

H.2 Changes Introduced in V1.4

The following describes the most significant changes which occurred in the NDF_ system between versions V1.3 and V1.4 (not the current version):

(1) The routines NDF_TUNE and NDF_GTUNE have been extended to support the new DOCVT tuning parameter. This allows automatic conversion of foreign format data files (SSN/20) to be disabled when not required.

(2) The maximum number of foreign data formats that can be recognised by the NDF_ library (SSN/20) has been increased from 20 to 50.

(3) Two new routines NDF_GTWCS and NDF_PTWCS have been provided to read and write World Coordinate System (WCS) information to an NDF. These WCS facilities are implemented using the new AST library (see SUN/210). NDF_GTWCS returns an AST pointer to a FrameSet and NDF_PTWCS expects a similar pointer to be supplied.

Note that this constitutes only a preliminary introduction of WCS facilities to the NDF library, mainly to permit the writing of data format conversion applications that support WCS information. Descriptions of the new routines are included in this document, but the main text does not yet contain an overview of the WCS facilities, for which you should consult SUN/210 at present. Further recommendations on the use of AST with the NDF_ library will be given in future, once experience with the new facilities has been gained.

(4) The new "WCS" component is now supported by other NDF_ routines, where appropriate (*e.g.* NDF_PROP, NDF_RESET, NDF_SCOPY, NDF_SECT and NDF_STATE).

(5) A bug has been fixed in the NDF_SBB routine which could occasionally cause it to access the bad-bits value for the wrong NDF.

(6) A bug has been fixed which could result in failure to access a named NDF data structure comprising one of the AXIS components of another NDF (for example, "NDF.AXIS(2)").

(7) The documentation (SUN/33 and SSN/20) has been updated to reflect these changes.

H.3 Changes Introduced in V1.5

The following describes the most significant changes which occurred in the NDF_ system between versions V1.4 and V1.5:

(1) An interface to the library has been added which is callable from C (see Appendices E and F).

(2) A new include file "ndf.h" has been added to support the C interface.

(3) Several new error codes have been introduced to support the C interface.

(4) References to the VMS operating system have been removed from the documentation (VMS is no longer supported by the current version of the NDF_ library).

(5) This document (SUN/33) has been updated to reflect recent changes to the library.

H.4 Changes Introduced in V1.6

The following describes the most significant changes which occurred in the NDF_ system between versions V1.5 and V1.6:

(1) Limited support for NDF array components stored in "scaled" form has been introduced. The new routine NDF_PTSZx will associate scale and zero values with an existing array component, thus converting it into a scaled array. See §12.4.

H.5 Changes Introduced in V1.7

The following describes the most significant changes which occurred in the NDF_ system between versions V1.6 and V1.7:

(1) NDF sections expressions can now include WCS axis values. See §16.3.

(2) New tuning parameters ("PXT...") can be used to supress the default propagation of named NDF extensions by NDF_PROP and NDF_SCOPY. See NDF_TUNE.

H.6 Changes Introduced in V1.8

The following describes the most significant changes which occurred in the NDF_ system between versions V1.7 and V1.8:

(1) The routine NDF_HSDAT has been added. This allows history records to be created with a specified date.

H.7 Changes Introduced in V1.9

The following describes the most significant changes which occurred in the NDF_ system between versions V1.8 and V1.9:

(1) A new standard WCS Frame called FRACTION has been introduced, This Frame is created automatically by the NDF library in the same that the GRID, PIXEL and AXIS Frames are created. The FRACTION Frame represents normalised pixel coordinates within the array - each pixel axis spans the range 0.0 to 1.0 in the FRACTION Frame.

(2) NDF sections specifiers can now use the "%" character to indicate that a value is a percentage of the entire pixel axis. Thus "m31(~50%,)" will create a section covering the central 50 percent of the NDF on the first pixel axis.

(3) A history component will now be added automatically to the NDFs created by NDF_CREAT and NDF_NEW if the "NDF_AUTO_HISTORY" environment variable, or "AUTO_HISTORY" tuning parameter, is set to a non-zero integer.

H.8 Changes Introduced in V1.10

The following describes the most significant changes which occurred in the NDF_ system between versions V1.9 and V1.10:

(1) A new function NDF_ISIN has been added, which determines if one NDF is contained within another NDF.

(2) The NDF_SCOPY and NDF_PROP functions now allow an asterisk to be used within the CLIST argument as a wild card to match all extension names.

H.9 Changes Introduced in V1.11

The following describes the most significant changes which occurred in the NDF_ system between versions V1.10 and V1.11:

(1) Limited support for NDF array components stored in "delta' compressed form has been introduced. The new routine NDF_ZDELT will create a delta compressed copy of an input NDF, and the new routine NDF_GTDLT will return details of the compression of a delta compressed NDF. See §12.5.

(2) A new function NDF_ZSCAL has been added, which creates a SCALED copy of an input NDF with SIMPLE or PRIMITIVE array components.

(3) A new function NDF_CREPL has been added, which allows an NDF placeholder to be created via a specified environment parameter.

H.10 Changes Introduced in V1.12

The following describes the most significant changes which occurred in the NDF_ system between versions V1.11 and V1.12:

(1) Support for 64 bit integer data values added.

(2) A new tuning parameter SECMAX has been aded, which allows the maximum number of pixels within an NDF section to be specified. See §23.3.

(3) A new routine NDF_CANCL can be used to cancel the association between an environment parameter and an NDF. This is identical to calling PAR_CANCL, except that NDF_CANCL provides an option to cancel all NDF parameters in a single call, without needing to know their names.

H.11 Changes Introduced in V1.13

The following describes the most significant changes that occurred in the NDF_ system between versions V1.12 and V1.13 (the current version):

(1) A new routine NDF_HCOPY has been added to copy history information from one NDF to another.

No changes to existing applications should be required, nor is any re-compilation or re-linking essential.

H.12 Changes Introduced in V2.0

The following describes the most significant changes that occurred in the NDF_ system between versions V1.13 and V2.0:

(1) The NDF library is now written entirely in C. However, the Fortran interface has not changed and is provided by a thin layer on top of the C library.

(2) The only change to the C interface is that there is now no need to call `ndfInit` to initialise the library. This is done automatically when an application first calls an NDF function.

(3) The C interface is now thread safe.

(4) New functions are provided in the C interface to allow each NDF to be locked for exclusive access by a single thread.

H.13 Changes Introduced in V2.1

The following describes the most significant changes that occurred in the NDF_ system between versions V2.0 and V2.1:

(1) A new tuning parameter called ROUND has been added that controls how floating-point values are converted to integer during automatic type conversion.

(2) The behaviour of ndfUnlock has changed. Previously, calling ndfUnlock would automatically annull all other identifiers associated with the same base NDF. This no longer happens. Such identifiers can now be used safely once the original thread has regained the lock on the base NDF.

H.14 Changes Introduced in V2.2

The following describes the most significant changes that occurred in the NDF_ system between versions V2.1 and V2.2 (the current version):

(1) It is now possible to specifiy the extent of an NDF section using arc-distance values (see §**??**).

Existing applications should be re-compiled and re-linked.